UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Optimización de la factorización de matrices no negativas en Bioinformática

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Edgardo Mejía Roa

Directores

Alberto Pascual Montano
Francisco Tirado Fernández

Madrid, 2016

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

TESIS DOCTORAL

Optimización de la Factorización de Matrices no Negativas en Bioinformática

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Edgardo M. Mejía Roa

Directores

Alberto Pascual Montano
Francisco Tirado Fernández

Madrid, 2015

Optimización de la Factorización de Matrices no Negativas en Bioinformática

Memoria presentada por Edgardo M. Mejía Roa para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de Alberto Pascual Montano y Francisco Tirado Fernández.

Madrid, 6 de noviembre de 2015.

A mi familia y a Carol.

Agradecimientos

Primeramente quisiera agradecer a mis directores de tesis, **Alberto Pascual Montano** y **Francisco Tirado Fernández**, por haberme permitido realizar este Doctorado, incluyendo la obtención de mi beca. Gracias por toda la ayuda en los trámites y en las publicaciones.

Así mismo, mis agradecimientos para **Manuel Prieto Matías**, quien ha sido mi mayor ayuda a lo largo de todo el doctorado (incluso desde antes, con el Proyecto de Fin de Carrera). Gracias por todas esas correcciones en los artículos, dudas varias, etc, etc.

También quisiera agradecer a Carlos García Sánchez, José Ignacio Gómez Pérez, Miguel Vázquez, Daniel Tabas, David Sánchez Foces, César Vicente, Enrique de la Torre, Pedro Carmona, Coral del Val, Javier Setoain, Rubén Nogales, Luis Canet, Ezequiel Lara Gómez, Juan Carlos Sáez Alcaide, Carolina Bonacic, Luis Piñuel y muchos otros, por su inestimable ayuda en esta tesis, tanto en cosas que abarcan parte de mi trabajo, como en pequeños detalles donde lo he necesitado.

En lo personal, mi primer pensamiento es para mi familia, especialmente para **mi madre y mi hermana**, que siempre me han apoyado y servido de referencia. También para **Carolina** y su *"toque de estilo"*, siempre dándome fuerzas para seguir. Sin ellas, nada de esto habría sido posible.

También a mis amigos, Bea, Edu, Pepe, Katia, Mary, Fernando, Joaquín y Arantxa, ayudándome y apoyándome en todo lo que podían.

¡¡¡GRACIAS A TODOS!!!

Financiación

Este trabajo se ha realizado con la financiación de la beca predoctoral ***Formación de Profesorado Universitario*** (***FPU***) del Ministerio de Educación y Ciencia, y de los

proyectos TIN2012-32180, TIN2005-05619, BIO2013-48028-R, BIO2007-67150-C03-02, CYTED-505PI0058, CSD00C-07-20811, CAM-P2010/BMD-2305 y CAM-P2006/-Gen-0166.

Índice general

Índice de figuras

Índice de cuadros

Resumen

En los últimos años se ha incrementado el interés de la comunidad científica en la ***Factorización de matrices no negativas*** (*Non-negative Matrix Factorization*, ***NMF***). Este método permite transformar un conjunto de datos de grandes dimensiones en una pequeña colección de elementos que poseen semántica propia en el contexto del análisis. En el caso de Bioinformática, NMF suele emplearse como base de algunos métodos de agrupamiento de datos, que emplean un modelo estadístico para determinar el número de clases más favorable. Este modelo requiere de una *gran cantidad de ejecuciones de NMF* con distintos parámetros de entrada, lo que representa una enorme carga de trabajo a nivel computacional.

La mayoría de las implementaciones de NMF han ido quedando obsoletas ante el constante crecimiento de los datos que la comunidad científica busca analizar, bien sea porque los tiempos de cómputo llegan a alargarse hasta convertirse en inviables, o porque el tamaño de esos datos desborda los recursos del sistema. Por ello, esta tesis doctoral se centra en la *optimización y paralelización de la factorización NMF*, pero no solo a nivel teórico, sino con el objetivo de ***proporcionarle a la comunidad científica una nueva herramienta*** para el análisis de datos de origen biológico.

NMF expone un alto grado de paralelismo a *nivel de datos*, de granularidad variable; mientras que los métodos de agrupamiento mencionados anteriormente presentan un paralelismo a *nivel de cómputo*, ya que las diversas instancias de NMF que se ejecutan son independientes. Por tanto, desde un punto de vista *global*, se plantea un modelo de optimización por *capas* donde se emplean diferentes tecnologías de alto rendimiento.

El *nivel inferior*, de menor granularidad, corresponde a los productos de matrices y otras operaciones algebraicas de las que se compone NMF. Un primer intento consistió en emplear la biblioteca de rutinas optimizadas de álgebra lineal, ***ATLAS*** (*Automatically Tuned Linear Algebra Software*). Posteriormente, se aprovecharon las capacidades de cómputo de las ***Unidades de procesamiento de gráficos*** (*Graphics Processing Unit*,

GPU). Para los primeros dispositivos programables se desarrolló una versión de NMF empleando el modelo de ***Procesamiento de flujos*** (*Stream Processing*) e implementada en `OpenGL` y `Cg`. Más adelante apareció otro paradigma, ***CUDA*** (*Compute Unified Device Architecture*), con el que se superó las diversas limitaciones de las versiones anteriores y se obtuvo una ganancia en tiempo muy importante respecto a la versión de base. No obstante, en el caso de dispositivos GPU discretos (normalmente dotados de una memoria específica de poca capacidad), las matrices de gran tamaño se deben procesar por bloques que deben previamente transferirse desde la memoria principal. La repercusión en el rendimiento de este tipo de operaciones suele ser bastante significativa.

Un *nivel intermedio*, de grano más grueso, puede establecerse en la distribución de los datos entre diversos elementos de cómputo (ej., un sistema multiprocesador o multi-GPU). Así, todos los elementos trabajan simultáneamente, cada uno aplicando alguna de las tecnologías del nivel inferior al subconjunto de datos que le es asignado. La implementación de este nivel se ha llevado a cabo aplicando a NMF un modelo de partición estática de dominio y utilizando la técnica del ***Paso de mensajes*** (*Message Passing Interface*, ***MPI***) para la sincronización de los diferentes elementos de cómputo. En un sistema multi-GPU (MPI+CUDA) se obtiene una ***ganancia superlineal*** respecto de un único dispositivo, ya que cada unidad debe ocuparse de un problema más pequeño, que no desborda su reducida capacidad de memoria y evita las continuas transferencias de datos.

Finalmente, el *nivel superior* corresponde a todas las instancias de NMF que son ejecutadas por los métodos de agrupamiento mencionados anteriormente. Tales instancias pueden distribuirse entre varios sistemas de cómputo (por ejemplo, entre diversos *clusters* de GPU), donde cada uno aplique las técnicas de los niveles inferiores. Para implementar este nivel, se desarrolló una infraestructura capaz de aprovechar las ventajas de las redes ***Grid***.

La nueva aplicación incluye una fase de preprocesamiento de matrices (por ejemplo, para eliminar valores negativos), así como una etapa posterior al análisis que genera una versión gráfica de los resultados. También dispone de interfaces web y de servicios web. Para facilitar aún más su utilización por parte de la comunidad científica se dispuso de una versión pública en línea, de uso gratuito y anónimo, a través de la dirección http://bionmf.dacya.ucm.es. Esta tuvo buena acogida por parte de la comunidad bioinformática, obteniendo una media de ***75 trabajos mensuales*** procedentes de distintos países.

En los últimos años ha crecido el interés de la comunidad científica, especialmente en

áreas de la Biomedicina, por diversas tecnologías de alto rendimiento tales como los procesadores de gráficos (GPU) o las redes *Grid*. Sin embargo, su utilización suele requerir conocimientos avanzados en informática y administración de sistemas. Por ello, esta tesis doctoral doctoral no se ha limitado a optimizar la factorización NMF desde un punto de vista teórico, sino también en construir una herramienta que verdaderamente sirva de ayuda al trabajo que a diario realizan los investigadores para extraer información biológica de enormes cantidades de datos experimentales. Si bien esta aplicación aún puede ser mejorada, supone una excelente herramienta que ayude a la comunidad científica en el Análisis Exploratorio de Datos.

Lista de palabras

Agrupamiento doble, Bioinformática, CUDA, Clasificación de muestras, Expresión génica, GPU, MPI, NMF, Redes grid, Servicios web.

Abstract

In the last few years, the ***Non-negative Matrix Factorization*** (***NMF***) technique has gained a great interest among the scientific community, since it is able to extract interpretable parts from high-dimensional datasets. In Bioinformatics, NMF is used as a basis for clustering methods that make use of a probabilistic model to compute the best suitable number of classes. Nevertheless, this model requires numerous executions of the NMF algorithm with different input parameters, which represents a considerable computing load.

A number of NMF implementations in different languages have been proposed in Bioinformatics and other fields of science, but their usage is limited by the large and constantly growing datasets that require analysis. Furthermore, the required processing time may become unpractical in many scenarios. Therefore, this Ph.D. dissertation is focused on the *optimization and parallelization of the NMF algorithm* with the aim to ***provide the Bioinformatics community with a new analysis tool*** for gene-expression data.

Similar to other linear-algebra algorithms, NMF exposes a high degree of *data-level parallelism* with different granularity. In contrast, the clustering methods expose computing-level parallelism, since the required instances of the NMF algorithm are totally independents. Therefore, a layer-based optimization model is proposed in this work, which uses different high-performance computing technologies.

The *lowest* or finest-grain level corresponds to matrix products and other algebraic operations that compose the NMF algorithm. A first approach was to make use of ***ATLAS*** (***Automatically Tuned Linear Algebra Software***), a library of optimized routines for algebraic operations. Another strategy was implemented through the high computing capabilities delivered by ***Graphics Processing Units*** (***GPU***). On legacy devices, a GPU-based version of the NMF algorithm was then developed following the ***Stream Processing*** paradigm. It was implemented on `OpenGL` and `Cg`. Afterward, a new GPU-computing model was employed, the ***Compute Unified Device Architec-***

ture (***CUDA***). This version of the NMF algorithm overcame most of limitations of the previous paradigm, and obtained a considerable speedup over the CPU-based version. However, on detached devices (typically equipped with a low-capacity on-board memory), large datasets must be explicitly blockwise transferred from the system's main memory to the GPU's memory, and processed accordingly. Such operations have severe negative effects on performance and reduce the speedup significantly.

A *medium*, coarser-grain, level can be established at the distribution of data among multiple computing elements. This way, all components work in parallel, each processing a different portion of the input matrix as described on the previous level. To implement this domain-decomposition scheme, the NMF versions described above were further parallelized by applying a static data partitioning model. Then, synchronization across different computing elements was ensured through the classic ***MPI*** (***Message Passing Interface***) technology. In a multi-GPU (MPI+CUDA) system, a ***super-linear speedup*** (compared to the single-GPU version) can be obtained when portions of input matrix assigned to each device are small enough to be transferred only once, at algorithm start.

Finally, the highest level represents the numerous NMF instances executed by the clustering methods. Such instances can be distributed among multiple remote computing systems (e.g., different GPU clusters). Each element then executes an instance of the NMF algorithm according to the lower levels of parallelism. To implement this level, the previous versions of NMF were further extended to make use of ***Grid Computing***.

The new application includes a preprocessing stage with different matrix transformation methods (e.g., to remove negative data on input matrix), and a post-processing step that generates some graphic outputs. *Web* and *Web Services* interfaces are also provided. Since its release as a public, anonymous, and free web-based tool at http://bionmf.dacya.ucm.es, the website has registered an average of ***75 jobs per month*** from several countries.

The Bioinformatics community is increasing the attention on different high-performance technologies, such as GPU or Grid computing. However, their usage often requires high expertise on computing sciences or system administration, even for simple tasks. This Ph.D. thesis has been focused, not only to optimize a widely used algorithm, NMF, but also to provide an application able to facilitate the daily work of researchers that are trying to extract biological information from hundreds of gigabytes of experimental data. Although this implementation can be further improved, it has become an important tool to assist life-sciences researches in the Exploratory Data Analysis process.

Keywords

Biclustering, Bioinformatics, CUDA, Gene Expression, GPU Computing, Grid Computing, MPI, NMF, Sample Classification, Web Services.

Capítulo 1

Introducción

Durante las últimas dos décadas, diversos campos de la Biología, como la Proteómica y la Genómica, han experimentado avances muy importantes gracias al desarrollo de nuevas técnicas automatizadas capaces de generar una gran cantidad de datos experimentales en lapsos de tiempo cada vez más reducidos. Esto, unido al progreso de las investigaciones en otros campos de la Biomedicina, ha dado lugar a la vasta acumulación de información generada por la comunidad científica sobre las moléculas y procesos básicos de la vida. Para poder analizar toda esta información, diversas disciplinas como la Estadística, la Minería de datos y la Informática han sido incluidas en las investigaciones en Biología y Biomedicina, lo que ha dado lugar a la aparición y desarrollo de nuevos campos de estudio, tales como la ***Bioinformática*** y la ***Biología Computacional***. Estas dos disciplinas están muy relacionadas en cuanto al uso de herramientas informáticas que faciliten las labores de investigación sobre determinados procesos biológicos (de hecho, ambos términos han llegado a utilizarse como sinónimos). No obstante, el objeto de sus análisis tienen una escala muy diferente: la Bioinformática se enfoca en los procesos que ocurren en las estructuras más básicas a nivel biológico (genes, células, proteínas, etc); por el contrario, la Biología Computacional es empleada en el estudio y simulación de grandes sistemas biológicos (órganos, organismos, sistema nervioso, etc).

A pesar de estos avances, en los últimos años el ritmo de producción de datos experimentales ha ido creciendo de manera exponencial [1, 2] debido, en parte, a la aparición de nuevas tecnologías que permiten generar en un único experimento la información equivalente a cientos de miles de pruebas tradicionales. Como consecuencia, la funcionalidad de las técnicas de análisis existentes ha ido quedando superada, generando así un cuello de botella en las tareas de investigación. Esto ha dado origen a la continua

búsqueda de tecnologías que permitan la producción de herramientas especializadas, y cada vez más eficientes, en el análisis de datos biológicos [3].

Un claro ejemplo de esta situación ocurre en la *Genómica Funcional* (disciplina orientada al estudio de los procesos empleados por los genes para modificar el estado del organismo); específicamente, en el ***Análisis de expresión génica*** mediante el uso de ***Chips de ADN*** (*DNA Microarrays*). Se trata de dispositivos capaces de cuantificar, de manera simultánea, el ***nivel de expresión*** o de *actividad* de un conjunto de genes. Su aparición y desarrollo ha supuesto un enorme salto cuantitativo en las tareas de investigación, ya que han permitido pasar de trabajar con genes individuales al estudio simultáneo de miles de genes e incluso de genomas enteros [4]. A continuación, se describe este proceso con más detalle.

1.1. Expresión génica

Un ***Gen*** es básicamente una sección de la cadena de ADN que describe o codifica una función determinada. Para llevar a cabo esta función, el gen es *transcrito* (copiado) a una cadena de *Ácido Ribonucleico Mensajero* (*ARNm*) que corresponde a su secuencia complementaria de moléculas de ADN. Posteriormente, el ARNm es llevado fuera del núcleo y es traducido a una cadena proteica. Este proceso de síntesis queda ilustrado en la figura 1.1. Es finalmente esta proteína, en su interacción con otras proteínas o redes proteicas, la que ejerce la funcionalidad determinada por el correspondiente gen. Se dice entonces que dicho gen se ha ***expresado***.

Dado que la mayoría de los procesos bioquímicos en la célula se realizan mediante interacciones entre proteínas, es posible caracterizar el estado celular y su actividad en el organismo a partir de la presencia de tales cadenas proteicas. De manera (casi) equivalente, también se puede determinar a partir del conjunto de genes que se expresan en un momento determinado [6].

Un método eficaz para medir el ***grado de expresión*** de un gen consiste en cuantificar los *niveles de ARNm* que han sido transcritos a partir de este. Como se mencionó anteriormente, gracias al desarrollo de los ***Chips de ADN*** se ha podido llevar esta labor a una escala superior, siendo posible analizar miles de genes de manera simultánea [4].

Es habitual realizar una cierta cantidad de experimentos para medir los niveles de expresión génica de un determinado organismo en diversas condiciones experimentales, distintas fases de desarrollo biológico, o utilizando muestras procedentes de diferentes tipos de tejido. Toda esta información permite constituir lo que se denomina ***perfiles***

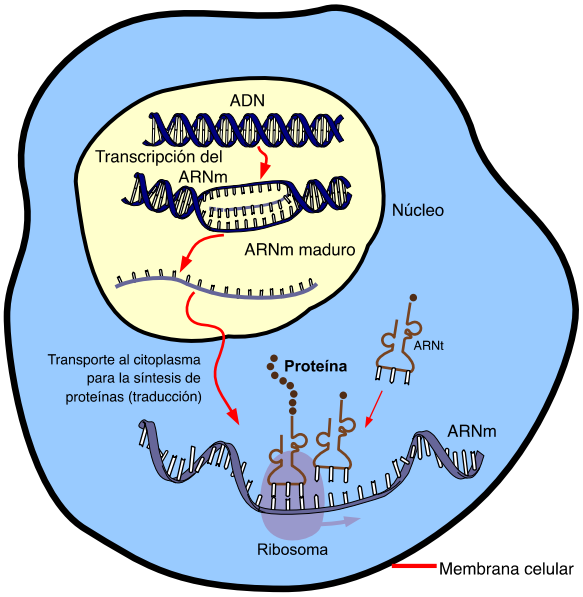

Figura 1.1: Síntesis de una proteína (cortesía de [5]).

Inicialmente se genera una cadena de *ARN Mensajero* (*ARNm*) que corresponde a la secuencia complementaria del gen. A continuación, un ribosoma recorre el ARNm cuando este último ya se encuentra fuera del núcleo. Por cada triplete de nucleótidos en el ARNm, se acopla un *ARN de transferencia* (*ARNt*) que aporta un aminoácido a la resultante cadena proteica.

de expresión génica. El análisis de este tipo de datos permite extraer información estadística de especial relevancia acerca de los diferentes estados celulares y su respuesta frente a diversos estímulos o condiciones externas. Por ejemplo, determinar qué genes se expresan de manera distinta en diversas condiciones experimentales (como podrían ser, tejido tumoral y tejido sano), permite obtener pistas que ayuden a centrar las investigaciones en ese subconjunto de genes y descartar el resto [7].

A continuación se describe el funcionamiento básico de los *Chips de ADN*.

1.1.1. Chips de ADN

Un Chip de ADN (*DNA Microarray*) [8] es una plataforma sobre la que se disponen secciones de ADN *de una sola hebra*. Tales fragmentos, que reciben el nombre de *sondas*, corresponden a un conjunto de genes conocidos de una determinada especie. Su ubicación en el chip es también conocida y tiene un diámetro del orden de micras, por lo que es posible albergar a cientos de miles de sondas distintas en una pequeña superficie de unos pocos centímetros cuadrados. La idea consiste en verter sobre dicha superficie cadenas de ARNm no conocidas de la muestra que se pretende estudiar. Como resultado, estas cadenas se unirán —en un proceso conocido como *hibridación*— exclusivamente con aquellas sondas que estén compuestas por su secuencia complementaria de molécu-

las de ADN. Por tanto, únicamente aquellas sondas hibridadas corresponderán a genes que se han expresado.

Para facilitar la detección de las sondas que hibridaron, las cadenas de ARNm son previamente marcadas con alguna molécula fluorescente. Además, se colocan del orden de millones de sondas del mismo tipo en cada posición del chip. De esta manera, las sondas en las que se ha producido hibridación reaccionarán a la luz a determinadas longitudes de onda. Todo este proceso queda ilustrado en la figura 1.2.

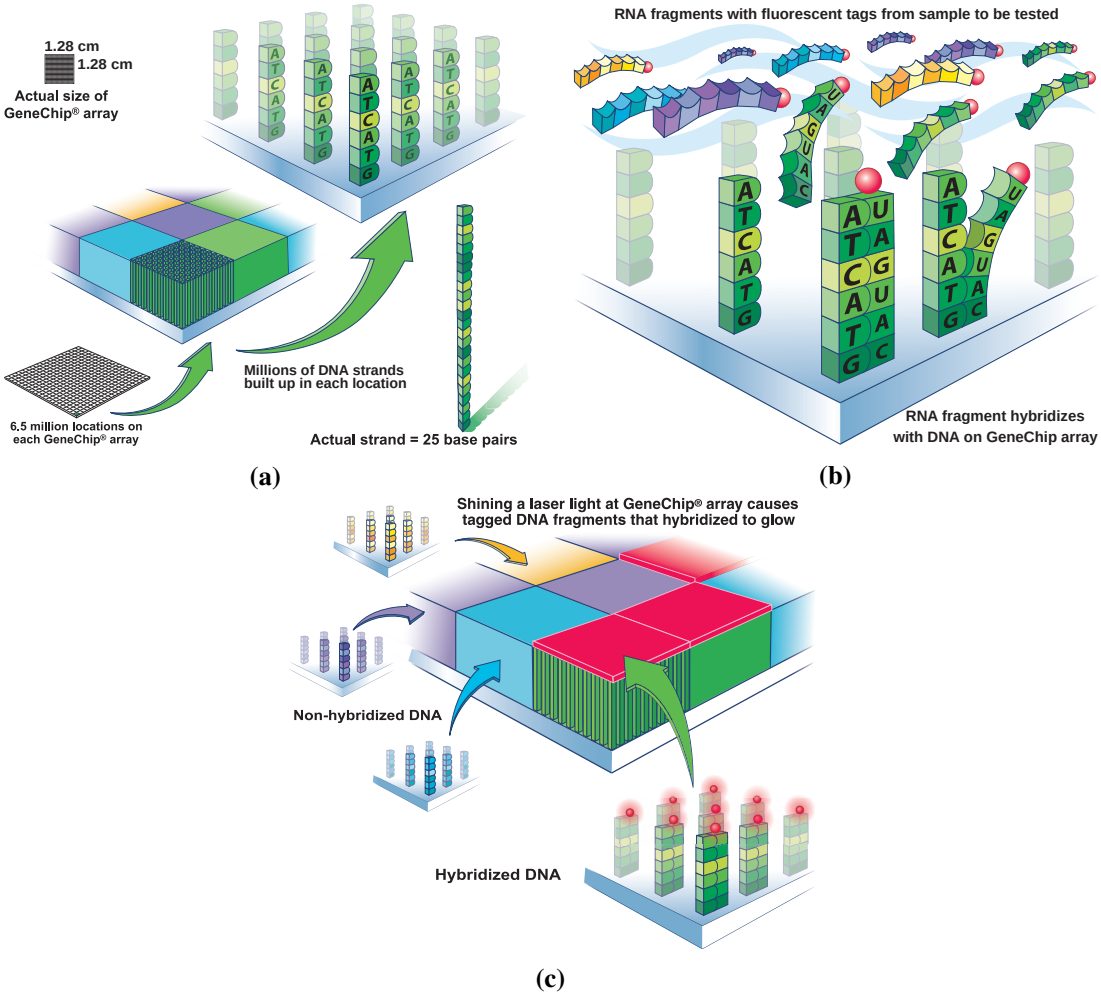

Figura 1.2: Funcionamiento de un Chip de ADN (cortesía de [9]).

(a) El chip contiene un conjunto de cadenas de ADN, de hebra simple, correspondientes a genes conocidos.
(b) Se marcan los fragmentos de ARNm de la muestra a analizar y se vierten sobre el chip, uniéndose con las secuencias que sean complementarias.
(c) Únicamente las sondas que contienen el marcador (en este caso, de color rojo) corresponden a genes que se expresaron.

Dependiendo del tipo de chip, justo antes del proceso de hibridación, el ARNm se mezcla con el procedente de una muestra de control para la que se utilizó un marcador

de color distinto. Como resultado, cada punto del chip mostrará una tonalidad diferente en función de la proporción de genes expresados en la muestra a estudiar respecto de la de control [8]. Por ejemplo, en un experimento donde la muestra problema tenga un marcador de color rojo y la de control uno verde: los puntos de rojos en el chip indicarán genes que se habían expresado en la muestra problema pero no en la de control, mientras que los puntos verdes corresponderán a la situación contraria. Por último, aparecerán puntos con un tercer color (por ejemplo, amarillo) en aquellos genes que no hayan variado sus niveles de expresión.
En otros tipos de chips, cada muestra se somete a un proceso de hibridación independiente y luego se superponen las imágenes resultantes.

Tras el proceso de hibridación, el chip es escaneado a fin obtener una imagen digital del resultado. Dicha imagen es posteriormente analizada para cuantificar la intensidad de la señal en cada una de las sondas. Los datos obtenidos suelen ser valores numéricos *crudos*, por lo que es habitual aplicar, a continuación, algún método de preprocesado y/o de normalización que permita adecuarlos a los métodos de análisis.

Entre los métodos de *preprocesado* más utilizados es posible destacar la transformación a escala logarítmica, para resaltar o atenuar las diferencias numéricas pero respetando las proporciones; la eliminación de genes que no hayan variado a través de distintas condiciones experimentales; la combinación de valores redundantes, en caso de que los haya; o el rellenado de valores ausentes, entre otros.
Por su parte, los métodos de *normalización* permiten mejorar la homogeneidad en los datos y eliminar aquellos valores producto de errores experimentales o de origen no biológico. Una revisión estos métodos se puede encontrar en [10].

1.2. Análisis de matrices de expresión génica

La información que se genera en un conjunto de experimentos con chips de ADN suele estar almacenada en una matriz numérica cuyas filas corresponden a genes y sus columnas representan diversas condiciones experimentales. Así, cada elemento $v_{i,j}$ indica el nivel de expresión del gen i en la condición experimental j, respecto al comportamiento del mismo gen en una muestra de control. Es decir, un valor positivo indica un grado de expresión superior al de la muestra de control, un dato negativo representa la situación opuesta y, por último, un cero denota una actividad similar.
Por otro lado, el vector correspondiente a la fila i constituye el ***perfil de expresión*** de ese gen, ya que contiene el conjunto de niveles de expresión obtenidos a lo largo de

todos los experimentos. Este mismo término se emplea de manera análoga para el vector asociado a la columna j, que contiene los perfiles de expresión de todos los genes en ese experimento.

Es habitual representar gráficamente esta matriz mediante un mapa de calor (*heat map*) en el que se emplea una escala de colores, tal y como se aprecia en la figura 1.3.

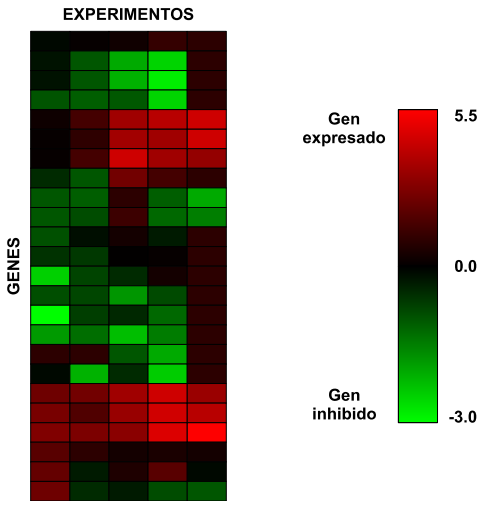

Figura 1.3: Ejemplo de una matriz de expresión génica

Las filas representan genes, y las columnas corresponden a distintas condiciones experimentales. Por tanto, cada elemento indica el nivel de expresión de un gen en una determinada condición experimental, respecto de una muestra de control. Este dato se representa de manera gráfica mediante una escala de colores (en este caso, verde-rojo).

Las dimensiones de este conjunto de datos suelen ser bastante importantes, pudiendo alcanzar decenas de miles de genes y varios miles de condiciones experimentales, lo que puede llegar a traducirse en cientos de millones de elementos. El análisis de una estructura de tal magnitud requiere la aplicación de metodologías estadísticas, de reconocimiento de patrones y de minería de datos. A continuación, se describe de manera muy breve algunas de las técnicas más empleadas.

1.2.1. Algoritmos de agrupamiento

Los algoritmos de agrupamiento (*Clustering*) se basan en distribuir los datos entre diversos conjuntos, agrupando aquellos elementos que presenten características comunes. En el caso particular del análisis de datos de expresión génica, se trata de identificar y agrupar genes (o condiciones experimentales) que presenten *perfiles de expresión* similares. Esta información es relevante para el proceso de análisis, ya que genes con patrones de expresión parecidos podrían estar implicados en los mismos procesos bioló-

gicos o estar regulados por los mismos mecanismos. En el caso de agrupar condiciones experimentales, podría indicar que están relacionadas con un mismo estado fisiológico (por ejemplo, muestras que proceden de un mismo tipo de tumor).

Entre los algoritmos de este tipo más utilizados se pueden destacar los siguientes:

1.2.1.1. Algoritmo de *k*-medias

Se trata de un método clásico de agrupamiento que distribuye los datos entre una cantidad predefinida de k grupos. El algoritmo consiste en conformar estos k conjuntos, e ir ajustándolos iterativamente de manera que cada elemento pertenezca al grupo más *cercano*. En términos matemáticos, este método intenta minimizar la siguiente función:

$$D = \sum_{i=1}^{k} \sum_{e \in G_i} \|e - c_i\|^2 \ ,$$

donde e es un elemento del grupo G_i y c_i es el *centroide* (media de los elementos) de dicho grupo.

El funcionamiento es el siguiente: dado un valor k, el algoritmo comienza distribuyendo los datos entre k grupos al azar. A continuación, se calcula el *centroide* de cada grupo y cada elemento es reasignado al conjunto cuyo centroide sea el más cercano, de manera que se generan k nuevos grupos. Este proceso se repite hasta que los centroides presenten pocas variaciones o se alcance un número máximo de iteraciones. El resultado es que elementos *cercanos* pertenecerán a un mismo grupo. La figura 1.4 resume este proceso.

A pesar de su rapidez y sencillez, este algoritmo requiere conocer con antelación el número de grupos (k) entre los que se deben distribuir los datos. Además, no existe ninguna garantía de alcanzar el resultado óptimo (correspondiente al mínimo *global* de la función), pudiéndose obtener tan solo un mínimo *local*.

1.2.1.2. Agrupamiento jerárquico

Este tipo de algoritmos intenta ordenar un conjunto de datos de manera jerárquica. Para ello, se emplea una estructura de árbol binario en la que se agrupan elementos que presenten un alto grado de similitud. Según esta organización, tales elementos quedan dispuestos en las *hojas* (nodos finales de la estructura), mientras que en la raíz se establece un representante del grupo. La figura 1.5 muestra un ejemplo de algoritmo

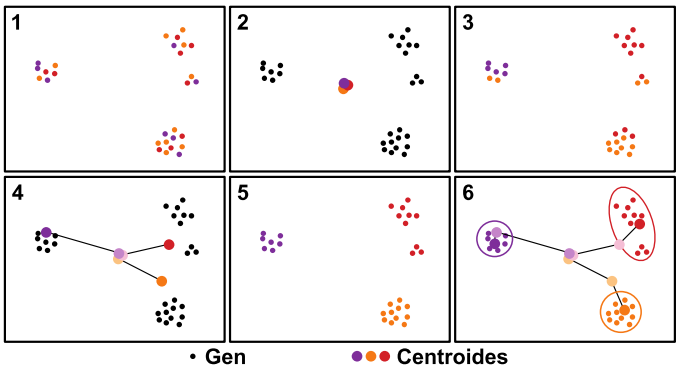

Figura 1.4: Algoritmo de *k*-medias (fuente: [11]).

Ejemplo de distribución de un conjunto de genes en $k = 3$ grupos:
(1) Distribución aleatoria de los genes.
(2 y 3) Se calcula el *centroide* (perfil promedio) de cada grupo y se asignan los genes al conjunto cuyo centroide esté más cerca.
(4 – 6) Se repiten los pasos 2 y 3 hasta que los centroides se estabilicen (se acerquen a la distribución de sus datos) o se alcance un máximo de iteraciones.

jerárquico aplicado a una matriz de expresión génica. La representación gráfica del árbol recibe el nombre de *dendrograma*.

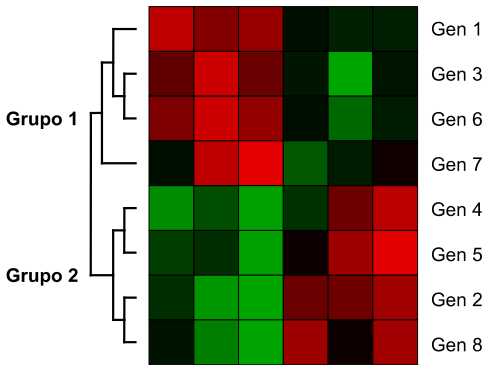

Figura 1.5: Agrupamiento jerárquico

Ejemplo de agrupamiento jerárquico aplicado a una matriz de expresión génica. Cada subárbol del dendrograma indica una agrupación de genes que presentan perfiles de expresión similares. Tales genes (filas de la matriz) quedan dispuestos en las hojas, mientras que la raíz corresponde al representante del conjunto. Nótese que las filas de la matriz han sido reordenadas para ilustrar mejor las agrupaciones.

Existen dos variantes de este algoritmo, descritas a continuación:

Aglomerativa

Esta variante comienza estableciendo conjuntos individuales de elementos. Se calculan entonces las distancias de todos contra todos y el resultado se guarda en una matriz numérica. A continuación, se seleccionan los dos elementos más cercanos, conformando un conjunto con ellos, y se recalcula toda la matriz de distancias. Este último paso se repite hasta obtener un único grupo.

Divisiva

En este caso, el algoritmo se inicia con un grupo que engloba a todos los elementos y, de manera iterativa, se van dividiendo en subconjuntos hasta alcanzar agrupaciones individuales.

Los algoritmos de agrupamiento jerárquico presentan entre sus ventajas la simplicidad del proceso, en el que se puede emplear una gran variedad de métricas de distancia, y la facilidad para representar gráficamente su resultado [12]. Sin embargo, tienen la desventaja de una cierta pérdida de similitud entre el representante de un grupo y los elementos que engloba, conforme crece el tamaño del conjunto. Además, cualquier error de asignación en un grupo se arrastra hasta el final del proceso [13].

1.2.1.3. Mapas auto-organizativos (SOM)

Un Mapa auto-organizativo (*Self-Organized Map* o *SOM*) [14] es una red neuronal capaz de proyectar un espacio multidimensional en uno de menor grado de dimensiones. Este mapa de neuronas, pretende tener una organización similar a la observada en el córtex cerebral de mamíferos y vertebrados.

En una red SOM, cada elemento de entrada es clasificado y asignado a la neurona del mapa que presente mayor similitud. No obstante, y a diferencia de otras redes neuronales, esta clasificación también afecta a las neuronas vecinas. Como resultado, neuronas que presenten similitudes quedarán dispuestas de forma agrupada en el mapa. Este proceso de organización del mapa se lleva a cabo de manera automática, sin supervisión externa, ya que está basado exclusivamente en las asociaciones encontradas entre las distintas entradas, y en las interconexiones neuronales.

Como se muestra en la figura 1.6, la red consiste en dos agrupaciones de neuronas. Una es la capa de entrada, ante la que se presenta cada dato en forma de vector $\boldsymbol{x}$. La otra, la capa de salida, suele tener una disposición bidimensional. Todas las neuronas de la capa de entrada están conectadas con las de salida. Igualmente, todas las de salida están conectadas entre sí. Tales conexiones tienen asociado ciertos valores (pesos) para reforzarlas o inhibirlas.

El funcionamiento de la red es el siguiente: (1) Cuando un vector $\boldsymbol{x}$ se presenta en la red, las neuronas de la capa de salida compiten entre sí (esto es, calculan su distancia —por lo general, euclídea— respecto de $\boldsymbol{x}$). (2) La neurona vencedora será aquella que obtenga el menor resultado (la más cercana). (3) Tanto la ganadora como sus vecinas modificarán sus pesos. Este proceso se repite hasta alcanzar algún criterio de

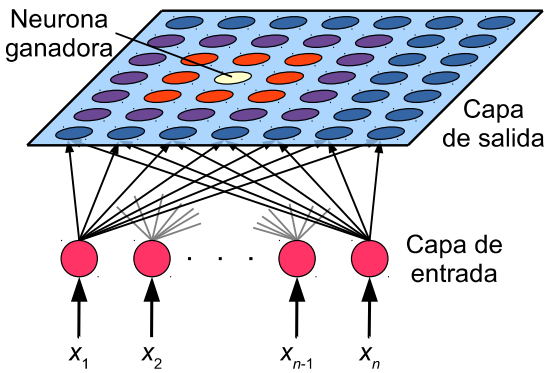

Figura 1.6: Mapa auto-organizativo (SOM)

Esquema de mapa auto-organizativo bidimensional. Ante cada dato de entrada (vector $\boldsymbol{x}$) las neuronas de la capa de salida compiten entre ellas. La ganadora (aquella que menos se distancie de $\boldsymbol{x}$), así como sus vecinas, actualizan sus pesos.

parada, por ejemplo, un número máximo de iteraciones o la estabilización de todas las neuronas.

Este tipo de algoritmo presenta entre sus ventajas la facilidad de visualización e interpretación de los resultados, así como la conservación de las relaciones topológicas y métricas entre los datos de entrada [15]. También destaca su relativa robustez frente al ruido de los datos, en comparación con otros algoritmos como el de K-medias. Por ello, los mapas auto-organizativos se han empleado en Bioinformática para el análisis de expresión génica [16, 17], así como para el reconocimiento de patrones y el análisis exploratorio de datos en otros ámbitos científicos [15].

No obstante, entre sus desventajas se encuentra el requisito de tener que especificar el tamaño y la estructura del mapa, lo que influye en la velocidad de aprendizaje. Además, si en el conjunto de entrada existe una cantidad importante de elementos con poca relevancia (por ejemplo, en el caso de expresión génica, genes con poca variación en sus perfiles de expresión), estos datos quedarán dispersos entre muchas neuronas, por lo que entorpecerán el resto de clasificaciones [18].

1.2.2. Métodos de factorización

Este tipo de algoritmos se basan en aplicar diversas operaciones matemáticas que transformen un conjunto de datos en entidades más manejables o que presenten algún otro tipo de ventaja. Una de estas transformaciones más habituales es la ***factorización***, que consiste en expresar un dato como una ***combinación lineal*** de otros elementos, denominados ***factores***, en la proporción indicada por unos determinados ***coeficientes***.

Es decir, representar un valor v de la siguiente manera:

$$v = c_1 f_1 + \cdots + c_k f_k = \sum_{i=1}^{k} c_i f_i \quad ,$$

donde f_i y c_i son, respectivamente, los k factores y coeficientes empleados en el proceso de aproximación.

En el caso de grandes conjuntos de datos, organizados de manera habitual en matrices, es posible extender esta idea definiendo adecuadamente los factores y coeficientes involucrados. Esto es, expresando todos los elementos de una matriz $\mathcal{M}_{n\times m}$ mediante las siguientes combinaciones lineales:

$$\begin{array}{lcl} v_{11} = f_{11}c_{11} + \cdots + f_{1k}c_{k1} & ; \quad \ldots \quad ; & v_{1m} = f_{11}c_{1m} + \cdots + f_{1k}c_{km} \\ \vdots & & \vdots \\ v_{n1} = f_{n1}c_{11} + \cdots + f_{nk}c_{k1} & ; \quad \ldots \quad ; & v_{nm} = f_{n1}c_{1m} + \cdots + f_{nk}c_{km} \end{array} \quad ,$$

ya que esto equivale a aproximar esta matriz mediante el siguiente producto:

$$\begin{bmatrix} v_{1,1} & \ldots & v_{1,m} \\ \vdots & \ddots & \vdots \\ v_{n,1} & \ldots & v_{n,m} \end{bmatrix} = \begin{bmatrix} f_{1,1} & \ldots & f_{1,k} \\ \vdots & \ddots & \vdots \\ f_{n,1} & \ldots & f_{n,k} \end{bmatrix} * \begin{bmatrix} c_{1,1} & \ldots & c_{k,m} \\ \vdots & \ddots & \vdots \\ c_{k,1} & \ldots & c_{k,m} \end{bmatrix} \quad ,$$

siendo el valor k, el denominado ***rango de factorización***. Nótese que en este ejemplo se han empleado dos matrices, pero los factores y/o coeficientes bien pudieran ser, a su vez, el resultado de otros productos de matrices.

Como se mencionó anteriormente, el objetivo de estas transformaciones consiste en que las nuevas entidades (en este último caso, matrices) tengan ciertas propiedades que faciliten su tratamiento matemático respecto del conjunto de datos original. Una de estas características es la ***reducción de la dimensionalidad*** que se obtiene cuando el número de elementos en las nuevas matrices es inferior a la cantidad de datos de entrada. En cualquier caso, el *rango de factorización* suele cumplir la siguiente condición:

$$k \leq \frac{m * n}{m + n} \tag{1.1}$$

Existe una gran variedad de métodos de factorización que son utilizados en diversas disciplinas científicas. En el caso de Bioinformática —y en particular, en el análisis de *matrices de expresión génica*— es posible destacar entre las metodologías más empleadas al *Análisis de Componentes Principales* (*PCA*) [19], la *Descomposición*

en Valores Singulares (*SVD*) [20, 21], el *Análisis de Componentes Independientes* (*ICA*) [22] y la *Factorización de Matrices no Negativas* (*NMF*) [23].

Todas estas técnicas tienen en común la posibilidad obtener una reducción de la dimensionalidad de los datos que, además, resalta la mayor varianza de los mismos. Es decir, que a diferencia de un simple *algoritmo de compresión*, las matrices obtenidas permiten destacar los patrones principales de los datos de entrada, facilitando así su análisis [24]. También se suelen utilizar como un paso previo a otros métodos de análisis, como por ejemplo, los *algoritmos de agrupamiento* descritos anteriormente.

No obstante, la mayor diferencia entre todos estos métodos de factorización, radica en la manera en que se calculan los factores y sus coeficientes. Por ejemplo, en el caso de la ***Factorización de Matrices no Negativas***, se establece la prohibición de emplear valores negativos. Como se verá en las siguientes secciones, esto ofrece la ventaja de representar los datos como la acumulación total o parcial de elementos con significado propio en el contexto del análisis. Por ello, este algoritmo se ha utilizado de manera mucho más extendida, en comparación con todos los métodos anteriores, tanto en Bioinformática como en otros campos de la ciencia.

1.3. La Factorización de Matrices no Negativas (NMF)

La factorización NMF (*Non-negative Matrix Factorization*) [23] es un método que permite transformar un conjunto de datos de grandes dimensiones en la *combinación lineal* de una pequeña colección de elementos que poseen *propiedades semánticas* en el contexto del análisis. Gracias a esta característica, se ha incrementado el interés por esta técnica en diversas áreas de investigación científica. Por ejemplo, en el reconocimiento facial y de objetos [23, 25–29], en estudios sobre el color [30, 31], en diferentes aplicaciones biomédicas [32–40], en el procesamiento de imágenes [41], en la transcripción de notas acústicas [42, 43], en el procesamiento de documentos [44–47], así como en otros métodos de procesamiento de señal [48–50].

En términos matemáticos, NMF se define como la descomposición de una matriz de entrada, $\mathbf{V} \in \mathcal{M}_{n\times m}(\mathbb{R})$, en otras dos matrices de menor tamaño, $\mathbf{W} \in \mathcal{M}_{n\times k}(\mathbb{R})$ y $\mathbf{H} \in \mathcal{M}_{k\times m}(\mathbb{R})$ (con $k \ll n, m$), cuyo producto se aproxima a la primera; es decir, $\mathbf{V} \approx \mathbf{WH}$. Esta operación equivale a representar cada elemento mediante la ***combinación lineal*** de un conjunto de k factores en la proporción indicada por otros tantos coeficientes.

Esto es,

$$(\mathbf{V})_{ij} \approx (\mathbf{WH})_{ij} = \sum_{q=1}^{k} w_{iq} h_{qj} ,$$

de manera que las matrices **W** y **H** almacenan, respectivamente, todos los factores y coeficientes necesarios para reconstruir todos los elementos de **V**.

Esta metodología debe su nombre a la restricción impuesta de que todos los valores deben ser ***mayores o iguales a cero***. Es precisamente esta condición la que permite obtener una representación de los datos de entrada basada en la ***acumulación de partes o secciones***, ya que coeficientes positivos convierten las combinaciones lineales en operaciones exclusivamente *aditivas*. En cuanto a los factores, estos aparecen como *componentes básicos con significado propio*, pues su estudio por separado (incluso una mera visualización) puede ser suficiente para obtener información relevante para el análisis [23].

Cuando se analiza una matriz en la que los datos de cada columna no son independientes entre sí, sino que conforman una misma señal (por ejemplo, píxeles de una misma imagen), los k factores obtenidos se corresponden con las columnas de la matriz **W**, por lo que reciben el nombre de ***vectores base***. Análogamente, las columnas de **H** se denominan ***vectores codificantes*** ya que se corresponden con las señales de la matriz de entrada. En este contexto, cada señal de la matriz de entrada queda expresada como la combinación lineal de todos los vectores base, en la proporción indicada por el correspondiente vector codificante [23].

Como se mencionó anteriormente, existen otros métodos de factorización similares, como por ejemplo, el *Análisis de Componentes Principales* (*PCA*) [19], la *Descomposición en Valores Singulares* (*SVD*) [20, 21] o el *Análisis de Componentes Independientes* (*ICA*) [22]. No obstante, en ellos no existe ninguna restricción respecto al signo de los datos, por lo que los factores obtenidos no aportan ningún tipo de información por sí solos. Un claro ejemplo de esta situación se muestra en la figura 1.7, donde se compara NMF y PCA.

1.3.1. El algoritmo de NMF

El algoritmo consiste en inicializar las matrices **W** y **H** con valores aleatorios *mayores o iguales a cero* para, a continuación, modificarlas iterativamente hasta que su producto se aproxime a **V**. Tales modificaciones, compuestas por diversas operaciones de álgebra lineal, se derivan de la minimización de alguna función objetivo que describa la *distancia*

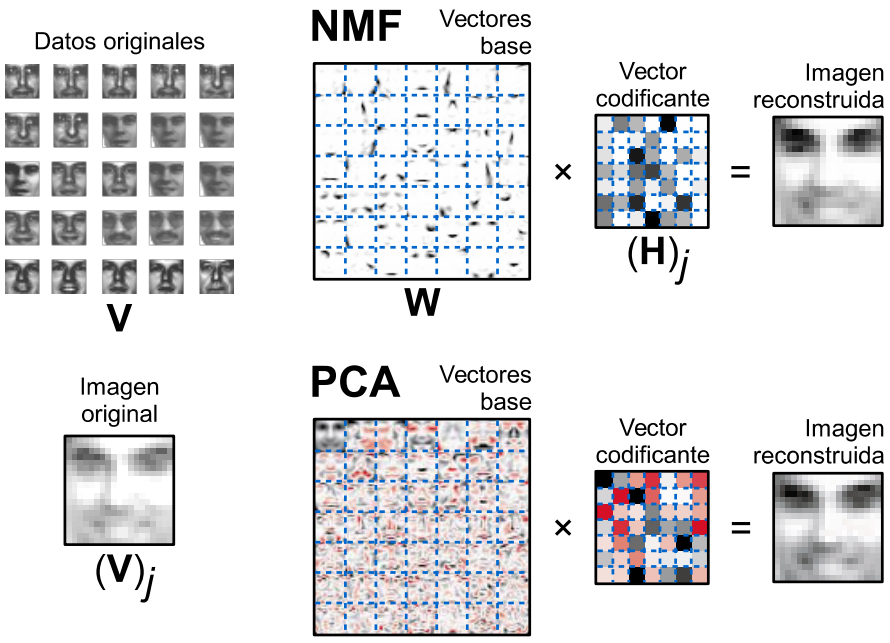

Figura 1.7: Ejemplo de factorización NMF. Comparación con PCA (fuente: [23]).

La matriz de entrada ($\mathbf{V}$) contiene distintas imágenes faciales, cuya organización real es una imagen por columna. Ambos algoritmos se ejecutan con un rango de factorización $k = 49$.

NMF: Los valores positivos se representan en color negro y los nulos en blanco. En la matriz $\mathbf{W}$ se obtienen vectores base que corresponden a diferentes partes de caras. Así, para reconstruir la imagen $(\mathbf{V})_j$ basta con "superponer" estas $k = 49$ imágenes base en la proporción indicada por el vector codificante $(\mathbf{H})_j$.

PCA: Se obtienen valores negativos (rojo) que, si bien se cancelan al reconstruir una imagen, a nivel de visualización no aportan ninguna información.

entre $\mathbf{WH}$ y $\mathbf{V}$. Existen diversas funciones de coste y métodos para minimizarlas, que conducen a diferentes *variantes* del algoritmo [23, 29, 49, 51–55]. A continuación se enumeran algunas de ellas:

NMF *"clásico"*

Este es el algoritmo original, propuesto en 1999, para el análisis de imágenes faciales [23]. Se basa en la minimización de la siguiente función objetivo:

$$F = \sum_{ij} \left[v_{ij} \log \left((\mathbf{WH})_{ij} \right) - (\mathbf{WH})_{ij} \right]$$

Esta función se obtiene al representar $\mathbf{WH}$ como una copia de $\mathbf{V}$ a la que se añaden pequeñas fluctuaciones, conocidas como *ruido de Poisson* o de *disparo*.

Las reglas que se derivan son las siguientes:

$$\boxed{h_{pj} \leftarrow h_{pj} \sum_i \left(w_{ip} \frac{v_{ij}}{(\mathbf{WH})_{ij}} \right)} \qquad \boxed{w_{ip} \leftarrow w_{ip} \sum_j \left(\frac{v_{ij}}{(\mathbf{WH})_{ij}} h_{pj} \right)} \tag{1.2}$$

$$\boxed{w_{ip} \leftarrow \frac{w_{ip}}{\sum_r w_{rp}}}$$

La última regla es un proceso de normalización en la matriz $\mathbf{W}$ que intenta eliminar posibles degeneraciones en $\mathbf{WH}$ [23].

NMF - Distancia euclídea

En este caso, la función objetivo a minimizar es la *Distancia euclídea* entre $\mathbf{V}$ y $\mathbf{WH}$ [51]:

$$\|\mathbf{V} - \mathbf{WH}\|^2 = \sum_{ij} \left(v_{ij} - (\mathbf{WH})_{ij}\right)^2 ,$$

de la que se derivan las siguientes reglas:

$$\boxed{h_{pj} \leftarrow h_{pj} \frac{(\mathbf{W}^T * \mathbf{V})_{pj}}{(\mathbf{W}^T * \mathbf{WH})_{pj}}} \qquad \boxed{w_{ip} \leftarrow w_{ip} \frac{(\mathbf{V} * \mathbf{H}^T)_{ip}}{(\mathbf{WH} * \mathbf{H}^T)_{ip}}}$$

NMF - Divergencia

Esta variante debe su nombre al empleo de la siguiente función objetivo [51]:

$$D(\mathbf{V}\|\mathbf{WH}) = \sum_{ij} \left[v_{ij} \log \left(\frac{v_{ij}}{(\mathbf{WH})_{ij}} \right) - v_{ij} + (\mathbf{WH})_{ij} \right] \tag{1.3}$$

Se trata de una adaptación de la *Divergencia de Kullback-Leibler* [56] utilizada normalmente para distribuciones probabilísticas. Es una función no negativa que se anula cuando $\mathbf{V} = \mathbf{WH}$. No obstante, al no ser simétrica (es decir, $D(\mathbf{V}\|\mathbf{WH}) \neq D(\mathbf{WH}\|\mathbf{V})$) no se trata de una "distancia", por lo que suele referirse como una *divergencia* desde $\mathbf{V}$ hasta $\mathbf{WH}$.

Las reglas de actualización que se obtienen son las siguientes:

$$\boxed{h_{pj} \leftarrow h_{pj} \frac{\sum_i \left(w_{ip} \frac{v_{ij}}{(\mathbf{WH})_{ij}} \right)}{\sum_i w_{ip}}} \tag{1.4a}$$

$$\boxed{w_{ip} \leftarrow w_{ip} \frac{\sum_j \left(\frac{v_{ij}}{(\mathbf{WH})_{ij}} h_{pj} \right)}{\sum_j h_{pj}}} \tag{1.4b}$$

Estas funciones de coste tienen el inconveniente de no ser convexas en **W** y **H** *simultáneamente* (solo en una de ellas a la vez), por lo que no se puede garantizar el poder encontrar el *mínimo global*. Sin embargo, sí es posible aplicar diversas técnicas que permitan alcanzar un *mínimo local*. Una de estas metodologías es el denominado ***Método del Gradiente Descendente***, que se describe a continuación.

1.3.1.1. Método del gradiente descendente (reglas aditivas)

Dada la superficie que conforma el espacio de búsqueda de una función multivariable, este método consiste en obtener una sucesión de puntos en la dirección de mayor "descenso" (pendiente negativa de la función) hasta alcanzar un mínimo local. Para ello, se parte de un cierto punto $\boldsymbol{x}_0$ y, en cada iteración, se calcula un nuevo punto que esté situado en la dirección marcada por el ***vector gradiente*** negativo del punto anterior. Es decir, actualizar cada punto mediante la siguiente ***regla aditiva***:

$$\boldsymbol{x}_{t+1} \quad \leftarrow \quad \boldsymbol{x}_t \, - \, \eta_t \nabla F(\boldsymbol{x}_t) \, , \tag{1.5}$$

para un cierto valor incremental η_t.

En el caso de la ecuación (1.3), la coordenada del punto correspondiente a la matriz **H** se actualizaría de la siguiente manera:

$$h_{pj} \quad \leftarrow \quad h_{pj} \, - \, \eta_{pj} \frac{\partial}{\partial h_{pj}} D(\mathbf{V} \| \mathbf{WH}) \tag{1.6}$$

Por tanto, se deriva respecto de **H**:

$$\frac{\partial}{\partial h_{pj}} D(\mathbf{V} \| \mathbf{WH}) \quad = \quad \sum_i w_{ip} \, - \, \sum_i \frac{w_{ip} v_{ij}}{\sum\limits_{q=1}^{k} w_{iq} h_{qj}} \quad ,$$

y se aplica la regla aditiva, intercambiando los signos, para obtener:

$$h_{pj} \quad \leftarrow \quad h_{pj} \, + \, \eta_{pj} \left[\sum_i \left(w_{ip} \frac{v_{ij}}{(\mathbf{WH})_{ij}} \right) - \sum_i w_{ip} \right] \quad , \tag{1.7}$$

siendo el coeficiente η_{pj} un valor *pequeño* para garantizar que la función $D(\mathbf{V} \| \mathbf{WH})$ decrece.

Con la matriz **W** se sigue un proceso similar, obteniendo la siguiente regla:

$$w_{ip} \quad \leftarrow \quad w_{ip} \; + \; \eta_{ip} \left[\sum_j \left(\frac{v_{ij}}{(\mathbf{WH})_{ij}} h_{pj} \right) - \sum_j h_{pj} \right] \tag{1.8}$$

La desventaja de este método es su lentitud para converger, especialmente cerca del mínimo local, donde se produce un cierto "zigzagueo". Otro método similar es el del *Gradiente conjugado*, que converge más rápido, pero suele ser mucho más difícil de implementar. Además, ambos métodos son bastante sensibles a la elección del valor incremental η, por lo que un valor demasiado alto deja de garantizar el decrecimiento de la función objetivo [51]. Es por ello que se suelen emplear, en su lugar, las denominadas ***Reglas multiplicativas***, descritas a continuación.

1.3.1.2. Reglas multiplicativas

Este tipo de reglas de actualización fueron propuestas por D. Lee y H. Seung en 2001 [51] en un intento de combinar las ventajas de los métodos del gradiente descendente y conjugado, obteniendo así un compromiso entre la facilidad de implementación y la velocidad de convergencia.

A diferencia de las reglas aditivas descritas anteriormente (en las que cada nuevo punto se obtenía sumando una cierta cantidad a las coordenadas actuales), en este método se obtiene el nuevo punto multiplicando el anterior por un determinado coeficiente que varía en función de la calidad de la aproximación. Esto es, un valor que va decreciendo conforme se minimiza la función, hasta alcanzar a la unidad.

Para calcular reglas multiplicativas a partir de las ecuaciones (1.7) y (1.8), los autores seleccionaron cuidadosamente los respectivos valores incrementales η:

$$\eta_{pj} = \frac{h_{pj}}{\sum_i w_{ip}} \qquad \text{y} \qquad \eta_{ip} = \frac{w_{ip}}{\sum_j h_{pj}} \quad ,$$

obteniendo así las ecuaciones (1.4) mostradas anteriormente.

1.3.1.3. Pseudocódigo

Teniendo en cuenta todos los aspectos anteriores, el algoritmo de factorización NMF puede resumirse en el pseudocódigo 1.1. En él, se emplean a modo de ejemplo las

ecuaciones (1.4), siendo muy similares las otras variantes. Los símbolos ".∗" y "./" denotan operaciones de matrices, de elemento a elemento.

Algoritmo 1.1 Factorización de Matrices no Negativas (NMF)

Precondiciones:

- $\mathbf{V} \in \mathcal{M}_{n\times m}(\mathbb{R})$ (con $n, m \geq 2$) contiene valores no negativos.
- Todas las filas y columnas de $\mathbf{V}$ contienen *al menos* un valor mayor que cero.
- $2 \leq k \leq \text{mín}(n, m)$

Postcondiciones:

- $\mathbf{W} \in \mathcal{M}_{n\times k}(\mathbb{R})$ y $\mathbf{H} \in \mathcal{M}_{k\times m}(\mathbb{R})$ contienen valores no negativos.
- $\mathbf{V} \approx \mathbf{WH}$

1: **Función** `NMF`($\mathbf{V}$, k)

2: `Inicializar`($\mathbf{W}$, $\mathbf{H}$) `// Valores aleatorios`

3: **Repetir**

4: $\boldsymbol{acc}_W \leftarrow$ `Reducir`($\mathbf{W}$) `// Vector fila`

5: $\mathbf{H} \leftarrow \mathbf{H} \;.*\; \left(\mathbf{W}^T\,[\mathbf{V} \,./\, (\mathbf{WH})]\right) \;./\; (\boldsymbol{acc}_W)^T$

6: $\boldsymbol{acc}_H \leftarrow$ `Reducir`($\mathbf{H}$) `// Vector columna`

7: $\mathbf{W} \leftarrow \mathbf{W} \;.*\; \left([\mathbf{V} \,./\, (\mathbf{WH})]\,\mathbf{H}^T\right) \;./\; (\boldsymbol{acc}_H)^T$

8: **Hasta Que** `Converge`($\mathbf{V}$, $\mathbf{W}$, $\mathbf{H}$)

9: **Valores de salida:** $\{\mathbf{W}, \mathbf{H}\}$

10: **Fin** `NMF`

El número de iteraciones realizadas variará en función de los datos de entrada, del punto de partida (valores iniciales de $\mathbf{W}$ y $\mathbf{H}$) y de la comprobación de *convergencia* que se realice. No obstante, en términos generales, el algoritmo puede llegar a ejecutar ***varios cientos de iteraciones***, lo que representa una ***importante carga computacional***, especialmente para matrices de gran tamaño.

1.3.2. NMF sin suavidad (nsNMF)

Como se comentó anteriormente, la capacidad de la factorización NMF para obtener una representación intuitiva de los datos que esté basada en partes o secciones, se debe

a la restricción de nunca emplear valores negativos. Sin embargo, esta condición no siempre es suficiente. Por ejemplo, en el caso de la figura 1.7, extraído de [23], existe un cierto grado de solapamiento entre los vectores base obtenidos, lo que contradice el concepto de representación por partes [25]. Por tanto, surge la necesidad de garantizar, de manera explícita, la ***raleza*** de los datos obtenidos; es decir, que estén más localizados y con menor grado de solapamiento. Con este objetivo fue propuesta la factorización ***NMF sin suavidad*** (***nsNMF***, del inglés *non-smooth NMF*) por Pascual-Montano *et ál.* en 2006 [57].

Existen otras variantes de NMF [27, 48, 58–62] que añaden restricciones a la función de coste (es decir, más operandos) para imponer raleza en la matriz **W**, en **H**, o en ambas. Por el contrario, nsNMF modifica todo el modelo del algoritmo incluyendo una nueva matriz, $\mathbf{S} \in \mathcal{M}_{k\times k}(\mathbb{R})$, con la que se puede controlar la raleza de manera explícita. Por tanto, la matriz de entrada, **V**, pasa a descomponerse como $\mathbf{V} \approx \mathbf{WSH}$, definiéndose esta nueva matriz, **S**, de la siguiente manera:

$$\mathbf{S} \;=\; (1-\theta)\mathbf{I} \;+\; \frac{\theta}{k}\mathbf{1} \quad ,$$

donde $\mathbf{I} \in \mathcal{M}_{k\times k}$ es la matriz identidad, $\mathbf{1} \in \mathcal{M}_{k\times k}$ es una matriz de unos, y el parámetro $\theta \in [0, 1]$ controla la suavidad del modelo.

Cuando θ vale cero, la matriz **S** es igual a la *matriz identidad*, por lo que no se impone ninguna suavidad y el modelo de descomposición es exactamente igual al habitual.
Sin embargo, a medida que θ tiende a 1, cualquier producto **XS** resultará en una matriz en la que todos los elementos de cada fila serán similares a la media de los valores de la correspondiente fila en **X**; y de manera análoga, a la media de sus columnas cuando la operación sea **SX**. Por ello, $\theta = 1$ corresponde a la *menor raleza* o mayor suavidad posible, ya que todos los elementos (de cada fila o columna) tendrán un mismo valor distinto de cero, en lugar tener algunos datos cercanos al cero y otros con valores muy superiores [57].

Las reglas de actualización consisten en una simple modificación de las ecuaciones (1.4), en las que se introduce la nueva matriz **S**. Además, se añade un tercer paso de normalización de la matriz **W** similar al empleado en la ecuación (1.2). Por tanto, las reglas

quedan de la siguiente manera:

$$\boxed{h_{pj} \leftarrow h_{pj} \frac{\sum_i \left((\mathbf{WS})_{ip} \frac{v_{ij}}{((\mathbf{WS})\mathbf{H})_{ij}} \right)}{\sum_i (\mathbf{WS})_{ip}}} \qquad \boxed{w_{ip} \leftarrow w_{ip} \frac{\sum_j \left(\frac{v_{ij}}{(\mathbf{W}(\mathbf{SH}))_{ij}} (\mathbf{SH})_{pj} \right)}{\sum_j (\mathbf{SH})_{pj}}} \tag{1.9}$$

$$\boxed{w_{ip} \leftarrow \frac{w_{ip}}{\sum_r w_{rp}}}$$

Finalmente, es importante acotar que el parámetro θ es totalmente *empírico*. En pruebas realizadas por el autor del modelo se ha encontrado que un valor de 0,5 sirve para obtener unos resultados razonables sin que la varianza explicada por el modelo se vea afectada drásticamente [57].

1.4. Uso de NMF para el análisis de datos de expresión génica

Como se mencionó anteriormente, la factorización de NMF ha tenido un gran interés en diversas áreas científicas, como la Bioinformática y otros campos de la Biomedicina. Por ejemplo, en minería de textos relacionados con la Biología [44, 46], en el análisis de secuencias proteicas [32], en la clasificación funcional de genes [63], en neurociencias [40, 64], así como en otras disciplinas *"ómicas"* (Genómica, Proteómica, Metabolómica, etc.) [37, 39]. Un repaso de algunos de estos y otros trabajos puede verse en [65].

No obstante, una de las mayores aplicaciones de NMF en Bioinformática, es en el ***Análisis de expresión génica*** [33–36, 38, 60, 66–71], descrito en la sección 1.1, cuyos datos suelen organizarse en una matriz de genes y experimentos (figura 1.3). En este caso, NMF divide genes y experimentos en k grupos que corresponden a alguna característica local o patrón en los datos. Por tanto, cada *vector base* o columna de la matriz $\mathbf{W}$, al tener la misma dimensión que un experimento (n filas), se denomina ***experimento base*** e indica el grado de pertenencia de cada gen a ese grupo. Análogamente, cada *fila* de la matriz $\mathbf{H}$ tiene la misma dimensión que un gen (m columnas), por lo que es denominada ***gen base***, e indica el grado de influencia que tiene el correspondiente grupo de genes en cada uno de los m experimentos. La figura 1.8 ilustra esta nueva interpretación.

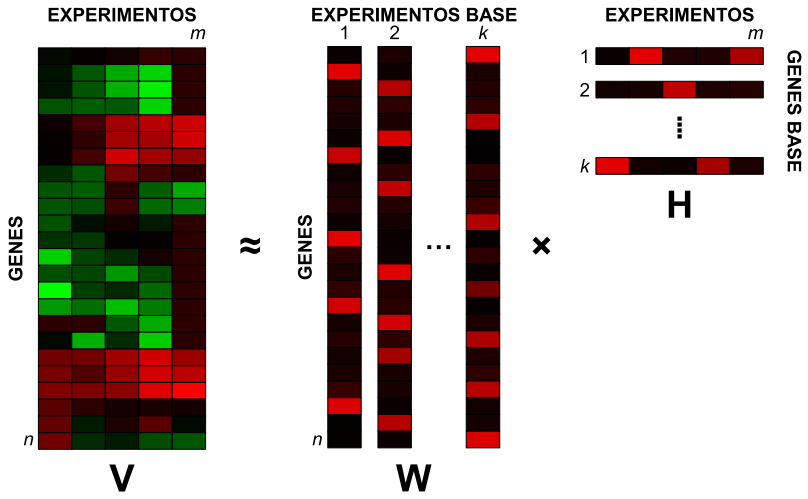

Figura 1.8: Aplicación de NMF al análisis de matrices de expresión génica

NMF divide genes y experimentos en k grupos correspondientes a algún patrón en los datos. Las columnas de la matriz **W** se denominan ***experimentos base*** e indican el grado de pertenencia de cada gen a cada grupo. Las *filas* de **H** se denominan ***genes base*** e indican el grado de influencia de un grupo en cada uno de los experimentos.

Un conjunto de genes con coeficientes relativamente altos dentro de un mismo *experimento base* recibe el nombre de ***módulo génico***, ya que indica que están relacionados con algún patrón en los datos. De manera análoga, un *gen base* en el que solo algunas condiciones experimentales contienen coeficientes mayores que cero, implica que tales experimentos están muy influenciados por alguna característica local en los datos. Por tanto, aquellos genes y condiciones experimentales que muestren valores altos en un *experimento base* y en el correspondiente *gen base*, indica que comparten similitudes locales en sus patrones de expresión génica, por lo que se les denomina *bicluster* o ***grupo doble de expresión génica***.

Este tipo de subconjuntos tienen especial relevancia en la extracción de información biológica, ya que es probable que ese grupo de genes estén implicados en los mismos procesos biológicos o estén regulados por los mismos mecanismos; y al mismo tiempo, es probable que esas condiciones experimentales estén relacionadas con un mismo estado fisiológico, como por ejemplo, muestras procedentes de un mismo tipo de tumor. Para detectar este tipo de estructuras, se suele emplear metodologías como la descrita a continuación.

1.4.1. Agrupamiento doble (*Biclustering*)

El Agrupamiento doble o *Biclustering* de genes y experimentos es una de las metodologías basadas en la factorización NMF que más se emplea en el análisis de datos de

expresión génica [35, 38, 39, 60, 72–74]. Este método obtiene k submatrices de datos (una por cada *factor*) donde quedan agrupados aquellos genes y experimentos con un alto grado de similitud local en sus patrones de expresión. Dado que tales estructuras no pueden ser detectadas mediante otros algoritmos de agrupamiento simples, como el *Agrupamiento jerárquico* o el de *k-medias* (ambos descritos en la sección 1.2.1), en los últimos años se han presentado diversos algoritmos en esta dirección. Algunos de ellos se encuentran recopilados en [75–77].

Uno de estos métodos es el propuesto por P. Carmona-Sáez *et ál*. en 2006 [35]. El algoritmo comienza aplicando la factorización NMF a la matriz de entrada; específicamente, se utiliza la variante *non-smooth NMF* (*nsNMF*) [57] (descrita en la sección 1.3.2) con el objeto de reforzar la raleza en los k *factores* obtenidos. A continuación, para cada *factor* (compuesto por una columna de **W** y la correspondiente fila de **H**), se seleccionan, por orden decreciente de sus coeficientes, aquellos genes y experimentos con valores más altos. Finalmente, se emplean los índices de fila y columna de estos grupos para extraer las correspondientes k submatrices del conjunto de datos de entrada. La figura 1.9 muestra el esquema general de la metodología.

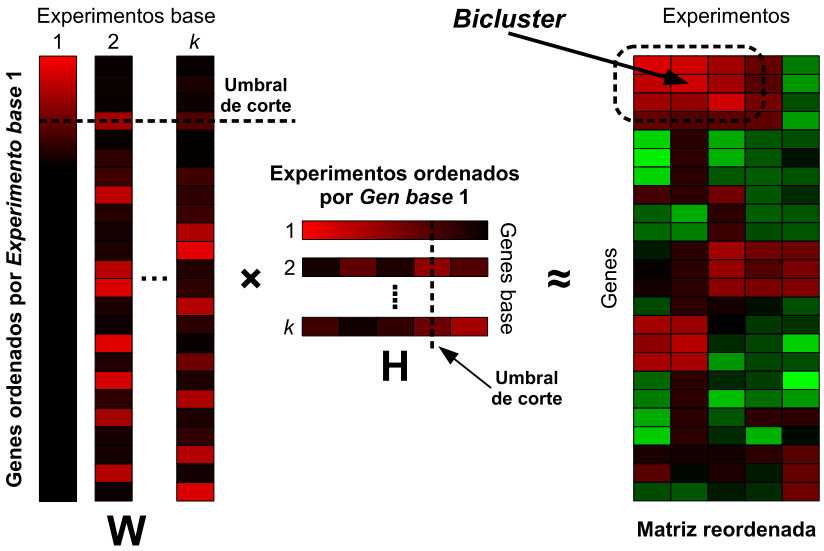

Figura 1.9: Algoritmo de agrupamiento doble (*Biclustering*) (fuente: [35]).

Este método reorganiza las matrices **W** y **H** por orden decreciente de los coeficientes del primer experimento y gen base, respectivamente. Con ello, se obtiene una permutación de filas y columnas de la matriz de entrada, en la que los elementos en la esquina superior izquierda corresponden a aquellos genes y experimentos que presentan mayor similitud en sus perfiles de expresión. Tal subconjunto de datos constituye un *bicluster*. Este proceso se repite para el resto de *factores* (columnas de **W** y filas de **H**), obteniendo otras $k - 1$ agrupaciones.

El criterio empleado para establecer los límites de cada *bicluster* consiste en considerar únicamente aquellos genes y experimentos que muestren coeficientes altos en ese grupo, y bajos en el resto. Esto es, al reordenar la matriz **W** en base a los coeficientes de una

de sus columnas, solamente se seleccionan los primeros genes que tengan su valor máximo en dicha columna, quedando el resto descartados. Análogamente, al reordenar la correspondiente fila de **H**, se eligen únicamente los primeros experimentos que tengan su valor máximo en dicha fila. Estas regiones de corte también aparecen ilustradas en la figura 1.9.
Otros trabajos han empleado estrategias diferentes, tales como la selección de un número predeterminado de genes [78], el uso de un valor fijo de umbral en los coeficientes [33], o la búsqueda de múltiples puntos de cambio [79].

Un aspecto importante, tanto para esta como para otras metodologías basadas en NMF, es el relativo a la ***naturaleza no determinista*** de este algoritmo de factorización. Esto es, dado que las matrices **W** y **H** se inicializan con ***valores aleatorios***, los resultados obtenidos pueden diferir de una ejecución a otra. En el caso del *Agrupamiento doble*, esto se traduce en la posibilidad de obtener variaciones en la asignación de genes y/o experimentos a cada *bicluster*.
Un método para solventar este problema consiste en repetir el algoritmo varias decenas de veces y analizar la *estabilidad* de los resultados obtenidos. Así, un conjunto de *biclusters* serán ***consistentes*** si pueden obtenerse sin sufrir (demasiadas) variaciones a lo largo de diversas ejecuciones, es decir, de manera independiente a las condiciones iniciales.
Para evaluar la *consistencia* de las agrupaciones obtenidas, primero se seleccionan las matrices **W** y **H** correspondientes a la *"mejor"* de las factorizaciones realizadas (es decir, aquella en la que $\mathbf{W} * \mathbf{H}$ se aproxime más a la matriz de entrada). A continuación, se analiza la repetición de las k agrupaciones obtenidas en esa factorización respecto del resto, considerándose *consistentes* cuando se encuentren en al menos el 80 % de los casos.

Otro elemento a considerar, de igual o mayor relevancia que el punto anterior, es el concerniente a la elección del ***rango de factorización*** (parámetro k). Emplear un valor muy bajo implica que cada *factor* contendrá mayor información —pues debe mantenerse la capacidad de reconstruir fielmente la matriz de entrada—, pero en consecuencia, los patrones hallados podrían ser demasiado genéricos y no aportar información útil. En el otro extremo, la utilización de un rango de factorización muy alto, aunque esté dentro de los límites habituales (ecuación (1.1)), podría conllevar a unos patrones demasiado detallados —o en un número demasiado elevado— que dificulten la extracción de información y su posterior análisis.
Encontrar un rango de factorización adecuado depende, en gran medida, del objetivo del análisis y del tipo de datos. No obstante, uno de los métodos más utilizados para estimar este valor es el propuesto en [34] para la *clasificación de muestras*. Esta técnica, descrita

a continuación, también aprovecha la naturaleza no determinista de NMF para evaluar la *estabilidad* de sus factorizaciones y, por consiguiente, la idoneidad del parámetro k empleado.

1.4.2. Clasificación de muestras (*Sample Classification*)

La Clasificación de muestras o *Sample Classification*, es un método propuesto por Brunet *et ál*. en 2004 [34] que utiliza la factorización NMF para agrupar aquellas condiciones experimentales (columnas de la matriz de entrada y de **H**) que presentan perfiles de expresión similares. Como se mencionó anteriormente, este tipo de información es útil ya que tales elementos podrían estar relacionados con un mismo estado fisiológico, como por ejemplo, pertenecer a un mismo tipo de tumor. Además, dado que el proceso de clasificación se basa únicamente en el pequeño conjunto de *meta-genes* o *genes base* (filas de **H**) obtenidos por NMF, esta técnica posee mayor robustez y precisión que otros métodos de agrupamiento simple que toman en cuenta todo el conjunto de genes. Este método de análisis utiliza la variante de *Divergencia* (ecuación (1.4)) ya que sus autores obtuvieron mejores resultados que con otras reglas de actualización [34].

No obstante, lo más relevante de esta metodología es la utilización de un modelo probabilístico para determinar el número adecuado de clases entre las que se pueden distribuir las condiciones experimentales. Es decir, este método permite ***estimar el rango de factorización*** (parámetro k) más conveniente y que mejor explica la varianza en los datos.

Para ello, el modelo aprovecha el comportamiento ***no determinista*** de la factorización NMF. Como se mencionó anteriormente, dado que las matrices **W** y **H** se inicializan con ***valores aleatorios***, los resultados obtenidos pueden diferir de una ejecución a otra. Esta propiedad es aprovechada para evaluar la ***estabilidad*** de las factorizaciones a lo largo de una serie de repeticiones del algoritmo, ya que si un rango de factorización es idóneo, es de esperar que sus resultados no varíen mucho de una ejecución a otra.

Para medir esa robustez, el algoritmo se basa en el ***Agrupamiento por consenso*** o *Consensus Clustering* [80] (razón por la que toda la metodología es referida con este nombre en algunas publicaciones [70, 71]), y además, propone el uso del ***Coeficiente de Correlación Cofenética*** (***CCC***) [81]. El método de agrupamiento permite obtener un aspecto visual de la calidad de las agrupaciones, mientras que el coeficiente proporciona un valor cuantitativo de las mismas. Todo el proceso de clasificación, por tanto, consiste en aplicar ambas técnicas a un grupo de rangos de factorización candidatos, seleccionando finalmente aquel donde el coeficiente CCC presente un valor pico.

En la figura 1.10 se muestra un ejemplo de *Clasificación de muestras* para distintos rangos de factorización.

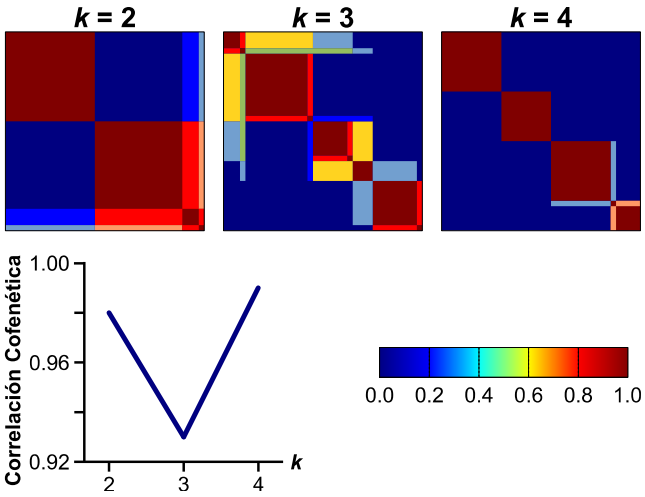

Figura 1.10: Clasificación de muestras (*Sample Classification*) (fuente: [34]).

Ejemplo de clasificación de muestras para tres rangos de factorización: $k \in \{2,3,4\}$. Para una mejor visualización, las matrices Consenso son reordenadas, agrupando las muestras que pertenecen a una misma clase. Los casos $k = 2$ y, especialmente, $k = 4$ representan factorizaciones relativamente estables, por lo que los valores de tales matrices aparecen próximos a 0 o a 1, y los respectivos coeficientes CCC contienen valores altos, cercanos a 1. Por el contrario, el caso de $k = 3$ corresponde a grandes variaciones en las factorizaciones, obteniendo así valores intermedios en la matriz Consenso y un coeficiente CCC inferior. De esta información se deduce que las muestras pertenecen a dos clases principales, que a su vez pueden dividirse en dos subgrupos (diferentes tipos de tumor, estado de desarrollo, etc).

1.4.2.1. Agrupamiento por Consenso (*Consensus Clustering*) y cálculo del Coeficiente de Correlación Cofenética (CCC)

Dado un cierto rango de factorización k, este método consiste en ejecutar el algoritmo de NMF una cierta cantidad de veces, analizando en cada una de ellas las k agrupaciones de experimentos obtenidas. Para ello, se emplea una ***matriz de conectividad***, $\mathbf{C} \in \mathcal{M}_{m\times m}(\{0,1\})$ (siendo m el número de condiciones experimentales), en la que cada valor c_{ij} es igual a 1 si las muestras i y j pertenecen a la misma clase, y 0 en caso contrario. De manera análoga al Agrupamiento doble de datos (*Biclustering*), se considera que cada experimento (columna de $\mathbf{H}$) pertenece a la clase o factor que corresponde a la fila (de $\mathbf{H}$) donde tiene su valor máximo. Por tanto, las muestras i y j pertenecerán a la misma agrupación si tales columnas de $\mathbf{H}$ tienen su valor máximo en la misma fila.
La matriz de conectividad también se utiliza en cada ejecución de NMF para comprobar la *convergencia del algoritmo*. Para ello, se evalúan los cambios producidos en esta matriz a medida que avanza el proceso de factorización, considerándose como finalizado

cuando transcurre una cierta cantidad de iteraciones sin que se haya percibido algún cambio.

El siguiente paso consiste en calcular la ***Matriz Consenso*** ($\overline{\mathbf{C}}$) que se define como la *media* de todas las matrices de conectividad obtenidas (una por cada ejecución de NMF), de manera que cada entrada tendrá un valor comprendido en el rango $[0,1]$ y reflejará la *probabilidad* de que dos experimentos (correspondientes a su fila y columna) pertenezcan a una misma clase. Por tanto, si las agrupaciones son estables, es de esperar que las distintas matrices de conectividad no varíen mucho y que todos los elementos de la matriz consenso tengan, entonces, valores muy próximos al cero o al uno.

A continuación, se aplica un algoritmo de agrupamiento jerárquico —como los descritos en la sección 1.2.1.2— a una de sus diagonales. Dado que los valores de esta matriz pueden interpretarse como el grado de *semejanza* entre todas las condiciones experimentales (tomadas de dos en dos), es posible utilizar su opuesta (es decir, $\mathbf{1} - \overline{\mathbf{C}}$) como medida de *distancia* para el algoritmo de agrupamiento jerárquico.

El Coeficiente de Correlación Cofenética (CCC) se define como la *Correlación de Pearson* (ρ) entre los elementos de un conjunto, representados por una matriz de distancias, y las distancias en el dendrograma obtenido al agrupar jerárquicamente dicho conjunto. El resultado es un número en el rango $[-1,1]$ donde el valor '1' indica una relación *directamente proporcional* entre ambos conjuntos, el '0' representa una falta de correlación *lineal* (aunque pudiera existir una *no lineal*), y '-1' implica una relación *inversamente proporcional*. Por tanto, el coeficiente CCC (*restringido al rango* $[0,1]$) proporciona una medida cuantitativa de la *calidad de un algoritmo de agrupamiento jerárquico*.
No obstante, en el caso del *Agrupamiento por Consenso*, el CCC proporciona una medida de la *dispersión* de $\overline{\mathbf{C}}$. Así, en una matriz con todos sus valores iguales a cero o a uno (valores muy dispersos), tendrá un coeficiente igual a '1'. Por el contrario, si sus valores se alejan de esos extremos (menor dispersión), entonces el coeficiente será bajo.

Todo este proceso del cálculo del CCC se repite para cada *rango de factorización* perteneciente a un *conjunto de valores candidatos*, resultando seleccionado aquel donde se obtenga un *valor pico* [34]. El esquema de la figura 1.11 resume toda la metodología de análisis. Nótese la ***gran cantidad de ejecuciones de NMF*** que se realizan.

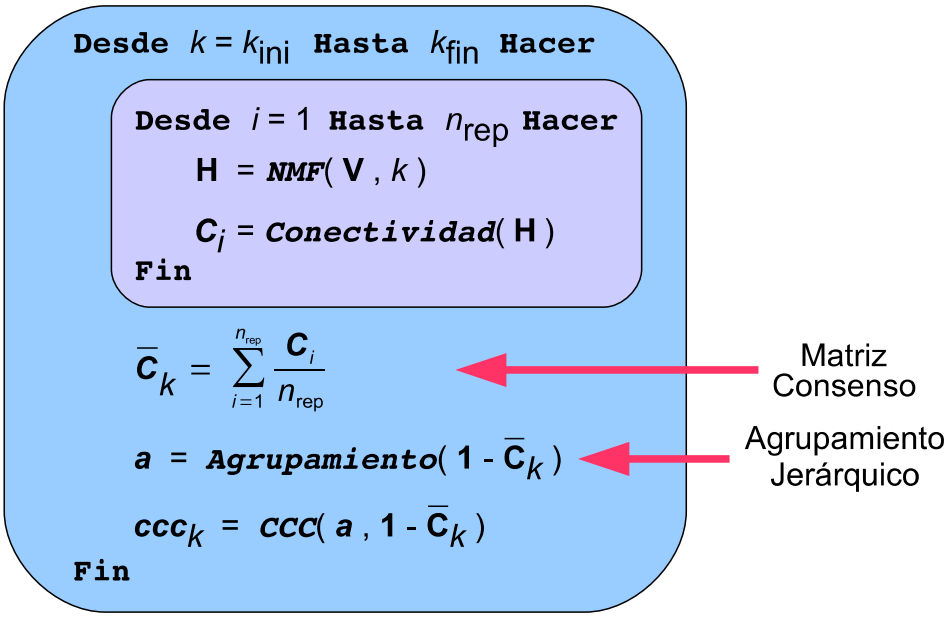

Figura 1.11: Esquema del método Clasificación de muestras (*Sample Classification*) [34]

Para cada rango de factorización candidato ($k \in [k_{\text{ini}}, \ldots, k_{\text{fin}}]$) se calcula la matriz de Consenso ($\overline{\mathbf{C}}_k$) y el correspondiente coeficiente CCC. Nótese que la primera operación implica n_{rep} ejecuciones de NMF *por cada valor que toma el parámetro* k. El rango de factorización *ganador* será aquel donde el CCC obtenga un valor pico.

Capítulo 2

Motivación y objetivos

Como se mencionó en el capítulo de Introducción, la ***Factorización de Matrices no Negativas*** (***NMF***) se ha convertido en los últimos años en uno de los algoritmos con mayor aplicación en Bioinformática para el análisis de datos de expresión génica. En este sentido, se han descrito dos metodologías —el ***Agrupamiento doble de datos*** (*Biclustering*) y la ***Clasificación de muestras*** (*Sample Classification*)— que emplean dicho algoritmo para la búsqueda de patrones y la reducción de los datos a analizar.

A pesar de las ventajas que ofrece la factorización NMF, la mayoría de las implementaciones propuestas hasta la fecha, tanto en Bioinformática como en otras áreas del conocimiento [34, 43, 45, 47, 82–89], han ido quedando obsoletas en rendimiento —o con una utilidad limitada— ante el constante crecimiento de los datos que la comunidad científica busca analizar, bien sea porque los tiempos de cómputo necesarios llegan a alargarse hasta convertirse en ***inviables*** o porque el tamaño de los datos ***desborda*** los recursos del sistema.

En efecto, como se describe en la sección 1.3.1 y se muestra en el pseudocódigo 1.1 de la página 18, el algoritmo de NMF está compuesto por diversas operaciones de álgebra lineal tales como productos de matrices o reducciones a vector. Estas operaciones —ya de por sí costosas a nivel computacional— pueden llegar a repetirse *miles de veces* hasta que el algoritmo converge a una solución aceptable. La situación es aún peor cuando se utiliza alguna de las dos metodologías de análisis mencionadas anteriormente, ya que estas se basan en ejecutar NMF *cientos de veces* con diferentes parámetros. Por tanto, en el caso de matrices de dimensiones moderadas o grandes, todo el proceso de análisis podría demorarse *días o semanas* de cómputo.

Este escenario queda ilustrado en la figura 2.1. Se trata de dos ejemplos del rendimiento del método *Clasificación de muestras*, obtenidos al ejecutar el código en `Matlab`

propuesto por los autores de la metodología [34]. En la primera prueba se utilizó una matriz (*DLBCL*, 44792×176) que contiene los niveles de expresión de casi cuarenta y cinco mil genes que fueron analizados en 176 muestras de tejido patógeno (*Diffuse Large B-Cell Lymphoma*) [90]. Por su parte, en el segundo ejemplo se emplea una matriz (*MCM*, 1961×8883) que corresponde a más de ocho mil imágenes, de 64×64 píxeles cada una, de un complejo *Mini-Chromosome Maintenance* (MCM) que procede de una *Methanobacterium Thermoautotrophicum* [91]. En ambos experimentos se quiso determinar el mejor *rango de factorización* entre un pequeño conjunto de *valores candidatos* ($\{2,\ldots,10\}$), repitiendo el proceso de factorización 40 veces por cada candidato. Nótese que estos parámetros implican un total de 360 ***ejecuciones de NMF*** en cada análisis.

Como se puede observar, los tiempos de cómputo con la matriz *DLBCL* son elevados, siendo ***inviables*** en muchos casos. La situación es aún peor con el otro conjunto de datos (*MCM*) y lo que aparece reflejado en la figura es una mera *estimación* del tiempo que podría demorar. Igualmente destaca su ***alto consumo de memoria***, este sí obtenido de manera empírica, que puede fácilmente desbordar los recursos de varios equipos informáticos de gama media.

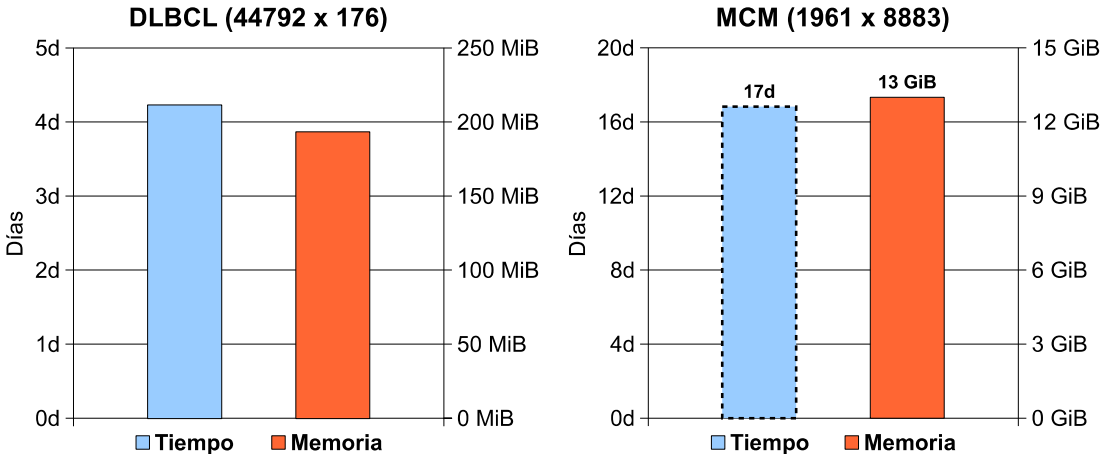

Figura 2.1: Rendimiento de *Clasificación de muestras* (implementación original [34]).

Tiempos de cómputo y consumo de memoria para dos matrices de grandes dimensiones. En ambas pruebas se ha intentado determinar el mejor rango de factorización en el intervalo $[2,10]$, repitiendo el algoritmo NMF 40 veces por cada valor candidato (360 ejecuciones en total). Los tiempos de cómputo obtenidos son excesivos, siendo el segundo caso una simple estimación. En esta última matriz también destaca su alto consumo de memoria (valor empírico), inviable en muchos equipos.
Plataforma de prueba: *Intel Xeon X5355*, 2,66 GHz, 16 GiB RAM y 8 GiB de espacio de intercambio. `Matlab` versión `7.5.0.338` (`R2007b`) [92].

En algunos casos, si el investigador tiene cierto conocimiento previo acerca de los datos, puede intentar procesar la matriz por partes de manera manual, aunque esto suele implicar una pérdida en la calidad del análisis. Sin embargo, la mayor parte de veces se

termina acudiendo a otros métodos de análisis, como los vistos en la sección 1.2.1, y asumiendo sus desventajas.

Tales circunstancias suponen un importante ***cuello de botella*** en las tareas de investigación, lo que impone la necesidad de buscar nuevas estrategias que permitan mejorar de manera importante el rendimiento de este algoritmo de factorización. Por ello, esta tesis doctoral se centra en la ***optimización de la factorización NMF*** mediante diferentes técnicas y tecnologías que han ido apareciendo en los últimos años.

No obstante, este trabajo no se restringe a la paralelización de este algoritmo desde un punto de vista teórico. Por el contrario, su objetivo es el de *proporcionar a la comunidad científica una nueva herramienta para el análisis de datos de expresión génica*. No hacerlo reduciría sustancialmente la utilidad de los resultados de este trabajo, pues el perfil medio de un investigador (bien sea en Biomedicina o en otras muchas áreas del conocimiento) no incluye conocimientos avanzados en programación ni en administración de sistemas.

Por tanto, para que la aplicación propuesta sea útil en un entorno *real* de investigación, debe tener las siguientes características:

- Incluir las dos *metodologías de análisis de datos de expresión génica* mencionadas anteriormente, así como algunos *métodos de preprocesamiento* que permitan normalizar y adaptar las matrices a los requerimientos de la factorización NMF.
- Permitir el procesamiento de conjuntos de datos *de gran tamaño* en equipos de *envergadura muy variada*: desde simples ordenadores portátiles y PC (supeditado a los recursos disponibles), hasta servidores y sistemas multiprocesadores de alto rendimiento.
- Debe tenerse en cuenta que *el usuario podría no tener conocimientos avanzados en informática*. De hecho, como es habitual en muchos laboratorios, personal técnico e investigador tienen perfiles de formación distintos, por lo que el usuario final bien podría no ser quien instale o configure la aplicación. Por tanto:
    - Debe existir una clara *distinción* entre la documentación para la instalación y la destinada a la utilización de la aplicación.
    - La herramienta propuesta debe incluir un *interfaz amigable* que oculte todos los detalles de implementación.
    - Debe ser robusta, que realice comprobaciones de todos los datos introducidos y que informe *de manera sencilla* los errores que se produzcan.

2.1. Implementación de referencia

Aunque en los últimos años se han ido proponiendo algunas implementaciones de NMF, este no ha sido el caso de los dos métodos de análisis. De hecho, aparte de las propuestas de sus respectivos autores [34, 35], la única herramienta conocida que los incluía al momento de comenzar esta tesis doctoral era la de Pascual-Montano *et ál*. ***bioNMF*** [84]. Se trata de una aplicación de tipo *standalone* (diseñada para su ejecución en *ordenadores personales*) que ofrece tres módulos de análisis correspondientes estas dos metodologías y a NMF. Entre sus características también se encuentra la disponibilidad de diversos *métodos de preprocesamiento* que permiten adaptar los datos de entrada a los requisitos de NMF, así como de *representación gráfica de los resultados*.
La publicación de esta aplicación tuvo muy buena acogida por la comunidad científica, pues cubría buena parte de las necesidades que tenían los investigadores de disponer de una herramienta de fácil utilización para el análisis de datos de expresión génica [84]. No obstante, su implementación en *Delphi 7* [93] para *Windows* (98, Me, 2000 o XP) fue alcanzando sus límites en prestaciones.

Dado que toda la *funcionalidad* que proporciona esta herramienta abarca gran parte de las metas planteadas para esta tesis doctoral, se utilizará como referencia a nivel de operatividad. Por tanto, este trabajo podría considerarse como una actualización de esta aplicación que permita el procesamiento de *matrices de gran tamaño* y en *diversos tipos de plataformas*.

2.2. Estructura de la tesis

El resto de este documento queda organizado de la siguiente manera: en el capítulo 3 se describe la funcionalidad de la aplicación *bioNMF* antes mencionada. A continuación, el capítulo 4 se examinan los algoritmos de la factorización NMF y de los métodos de análisis *Clasificación de muestras* y *Agrupamiento Doble*, así como sus posibilidades de optimización. Posteriormente, el capítulo 5 detalla todo el modelo de paralelización aplicado a los algoritmos, así como las tecnologías empleadas para ello. Además, se muestran los resultados de pruebas de rendimiento realizadas. Seguidamente, el capítulo 6 describe los diversos interfaces implementados para la aplicación. Finalmente, el capítulo 7 expone las conclusiones, enumera las contribuciones realizadas y propone algunas líneas de trabajo futuro.

Es importante resaltar que gran parte de las contribuciones que han dado lugar a esta

tesis doctoral fueron publicadas en revistas del área de ***Bioinformática***, en las que muchos de sus lectores tienen conocimientos muy básicos en Informática (únicamente a nivel de usuario). Esto ha obligado a ***adaptar el lenguaje*** reduciendo al mínimo la terminología técnica, así como a ***simplificar los detalles*** de la implementación y del funcionamiento de las tecnologías empleadas. Este enfoque se *ha mantenido* en la Memoria de esta tesis doctoral, por lo que se ha *empleado un lenguaje más neutro* y se ha *resumido* gran parte de los conceptos informáticos empleados habitualmente.

2.3. Contribuciones

Entre las contribuciones a las que ha dado lugar este trabajo, se encuentran:

- **REVISTAS** (por orden cronológico descendente):
  - ***NMF-mGPU: Non-negative Matrix Factorization on Multi-GPU systems***. Edgardo Mejía Roa, Daniel Tabas Madrid, Javier Setoain, Carlos García, Francisco Tirado y Alberto Pascual Montano. *BMC Bioinformatics*, 02/2015; **16**:43–55. DOI:10.1186/s12859-015-0485-4 http://www.biomedcentral.com/1471-2105/16/43

    En este trabajo se propone una implementación de la factorización NMF que hace uso de uno o varios Procesadores Gráficos (GPU). Este tipo de dispositivos ha despertado el interés de la comunidad científica, incluyendo al área de Bioinformática, debido a la enorme potencia de cálculo que tienen para operaciones de álgebra lineal. El código, de acceso libre, se encuentra disponible en la dirección: http://bioinfo-cnb.github.io/bionmf-gpu/ y puede ser ejecutado en sistemas de diversa envergadura, desde simples ordenadores portátiles hasta sistemas multi-GPU de alto rendimiento.

  - ***bioNMF: a web-based tool for Nonnegative Matrix Factorization in Biology***. Edgardo Mejía Roa, Pedro Carmona Sáez, Rubén Nogales, César Vicente, Miguel Vázquez, Xiao Y. Yang, Carlos García, Francisco Tirado y Alberto Pascual Montano. *Nucleic Acids Research*, 06/2008; **36** (2, *Web Server issue*): W523–W528. DOI:10.1093/nar/gkn335. http://nar.oxfordjournals.org/cgi/content/full/36/suppl_2/W523

    En este trabajo se propone una herramienta en línea para el análisis de datos de expresión génica. La aplicación, de acceso libre y anónimo, y que se ejecuta en servidores del grupo, obtuvo buena acogida en la comunidad

Bioinformática, llegando a tener una media de 70 trabajos mensuales procedentes de diversos países del mundo. Todavía se encuentra disponible en la dirección: http://bionmf.dacya.ucm.es

- **CONGRESOS INTERNACIONALES** (por orden cronológico descendente):

  - ***Parallelism on the Nonnegative Matrix Factorization***. Edgardo Mejía Roa, Carlos García, José Ignacio Gómez, Manuel Prieto, Christian Tenllado, Alberto Pascual Montano y Francisco Tirado. *"Applications, Tools and Techniques on the Road to Exascale Computing"*, vol. **22** de *Advances in Parallel Computing*; 05/2012:421–428; *IOS Press BV*. DOI:10.3233/978-1-61499-041-3-421 http://ebooks.iospress.nl/volumearticle/26557

    En este trabajo se analiza el rendimiento de una versión preliminar de la implementación de la factorización NMF para Procesadores Gráficos (GPU), tomando en cuenta distintos paradigmas de programación (CUDA, MPI y MPI+CUDA) y las limitaciones de cada uno de ellos.

  - ***Biclustering and classification analysis in gene expression using Nonnegative Matrix Factorization on multi-GPU systems***. Edgardo Mejía Roa, Carlos García, José Ignacio Gómez, Manuel Prieto, Francisco Tirado, Rubén Nogales, Alberto Pascual Montano. *Proceedings of the 11th International Conference on Intelligent Systems Design and Applications* (*ISDA*), 2011: 882–887; *IEEE Press*. DOI:10.1109/ISDA.2011.6121769 http://doi.ieeecomputersociety.org/10.1109/ISDA.2011.6121769

    En este trabajo se analiza el uso de procesadores gráficos (GPU) para implementar la factorización NMF, teniendo en cuenta que este algoritmo sirve de base para los métodos de análisis de datos de expresión génica: Agrupamiento doble (*Biclustering*) y Clasificación de Muestras (*Sample Classification*).

  - ***bioNMF-grid: An online grid-based tool for Non-negative Matrix Factorization in Biology***. Edgardo Mejía Roa, Miguel Vazquez, Pedro Carmona Sáez, Carlos García, Francisco Tirado, Alberto Pascual Montano. *Proceedings of the 2nd EELA-2 Conference*; 2009: 133–139; Editorial CIEMAT. ISBN:978-84-7834-627-1 http://publicacionesoficiales.boe.es/detail.php?id=012847109-0001

    En este trabajo se propone una herramienta en línea para el análisis de datos de expresión génica. La aplicación, de acceso libre y anónimo, utiliza la

infraestructura *grid* para ejecutar los trabajos cuando el *cluster* local del grupo de investigación se encuentra en uso.

Capítulo 3

bioNMF: un software para factorización no negativa de matrices en Biología

En este capítulo se describe la aplicación ***bioNMF*** desarrollada por Pascual-Montano *et ál.* [84] en 2006, cuyas características servirán de referencia *a nivel de operatividad* para los objetivos de esta tesis.

3.1. Funcionalidad

El menú principal de la aplicación de *bioNMF* está dividido en tres módulos: carga de datos, transformación de datos y módulos de análisis (ver figura 3.1).

Lectura de datos: *bioNMF* acepta matrices de datos contenidas en ficheros de texto plano con o sin columnas y filas de encabezamiento.

Transformación de los datos: *bioNMF* ofrece siete métodos distintos para la normalización de los datos. Además, si los datos contienen valores negativos, por ejemplo, han sido previamente transformados a escala logarítmica, la aplicación ofrece distintas estrategias para acomodarlos al requisito de no negatividad:

- *Restar el mínimo absoluto*: El valor mínimo es restado a todos los elementos de la matriz.
- *Duplicar los datos por filas*: Esta aproximación fue usada por Kim y Tidor para el análisis de datos de expresión génica [**?**]. Cada fila es representada

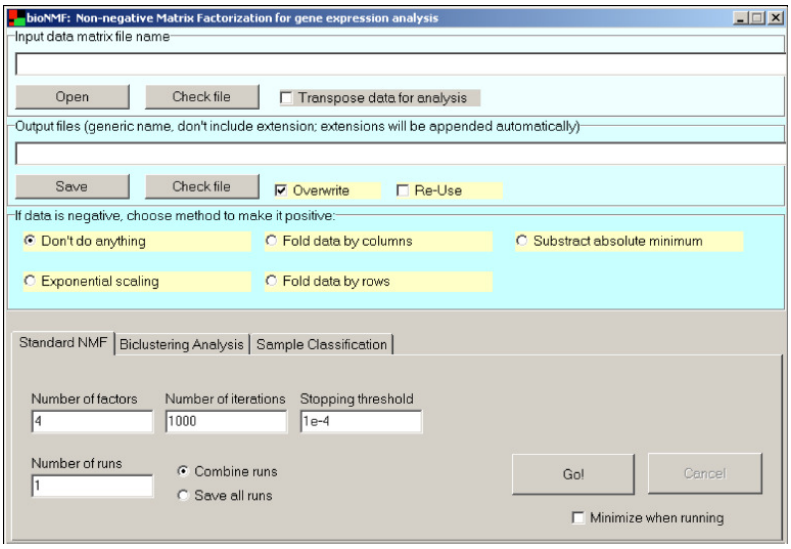

Figura 3.1: Menú principal de *bioNMF*

La herramienta está dividida en tres módulos principales: cargado de datos, preprocesamiento y módulos de análisis (NMF estándar, análisis de agrupamiento doble y clasificación de muestras).
Fuente: [84].

en dos filas en una nueva matriz, la primera es usada para indicar expresión positiva y la segunda para indicar expresión negativa. Este proceso duplica el número de filas de los datos originales.

- *Duplicar los datos por columnas*: De una forma similar a la anterior, esta opción permite al usuario transformar los datos a valores positivos duplicando el número de columnas. Cada columna es representada en dos nuevas columnas, la primera indica expresión positiva y la segunda expresión negativa. En este caso se duplica el número de columnas de la matriz original.

- *Escalado exponencial*: Los datos son escalados exponencialmente para hacerlos positivos. Esta es la operación inversa a la transformación logarítmica.

Análisis de los datos: *bioNMF* ofrece tres módulos diferentes de análisis que engloban las aplicaciones más importantes de NMF en el análisis de expresión propuestas hasta la fecha:

- NMF estándar.

- Agrupamiento doble.

- Clasificación de muestras.

A continuación se describen en detalle los tres principales módulos de análisis implementados en *bioNMF*.

3.1.1. Módulo de NMF estándar

Este módulo (ver figura 3.2) aplica el modelo estándar de NMF propuesto por Lee y Seung [23] a cualquier tipo de matriz de datos. La naturaleza exacta de los datos y el post-procesamiento de los resultados obtenidos en la factorización depende de la aplicación particular y del objetivo último del análisis.

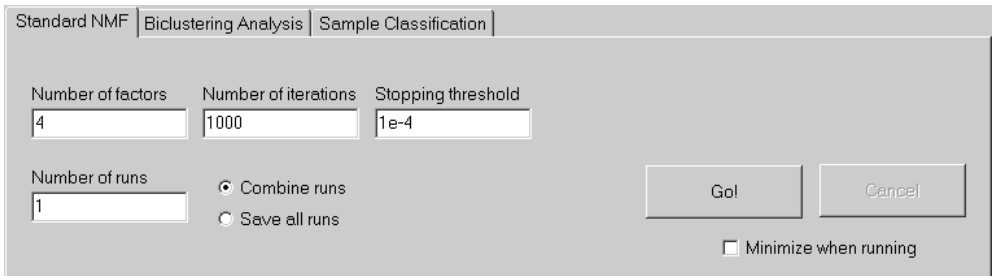

Figura 3.2: Panel del módulo de NMF estándar

Los parámetros necesarios para la ejecución son: i) *Number of factors*: Número de factores. ii) *Number of iterations*: Número máximo de iteraciones del algoritmo. iii) *Stopping threshold*: Cuando los cambios entre las iteraciones de **W** y **H** son menores que este valor el algoritmo finaliza (incluso si el número de iteraciones máximas no ha sido alcanzado). iv) *Number of runs*: Número de ejecuciones que se llevaran a cabo con distintas condiciones iniciales. Los resultados se pueden salvar independientemente seleccionando *"Save all runs"* (los ficheros de salida para cada factorización tendrán una etiqueta *"-RandomRunX"*, donde *X* corresponde al número de la factorización), o se pueden combinar en un único fichero seleccionando *"Combine runs"*.
Fuente: [84].

3.1.2. Módulo de agrupamiento doble

Este módulo (figura 3.3) implementa la metodología de *agrupamiento doble* de datos (*biclustering*), basada en la variante de NMF sin suavidad *nsNMF* (descrita en la sección 1.3.2).

3.1.3. Módulo de clasificación de muestras

Este módulo (figura 3.4) implementa la metodología propuesta por Brunet *et ál.* [34] para la clasificación de experimentos utilizando sus perfiles de expresión. En este caso se emplea las ecuaciones de NMF de *Divergencia*, descritas en la sección 1.3.1.

Un ejemplo de los resultados puede verse en la figura 3.5.

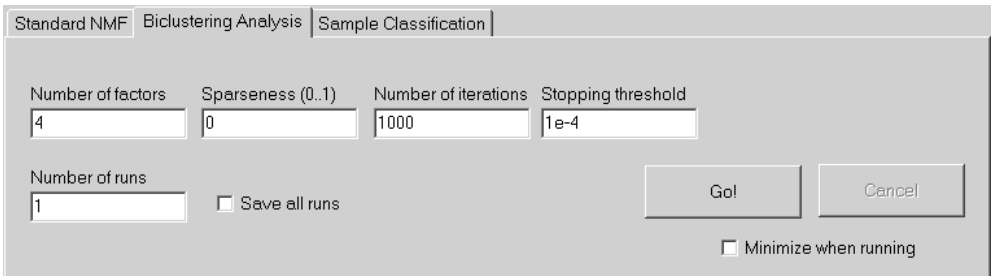

Figura 3.3: Módulo de agrupamiento doble

Además de los parámetros compartidos con el módulo de NMF estándar, este módulo incluye el parámetro para controlar el nivel de suavidad de la matriz **W** y **H** (*Sparseness* [0 . . . 1]). Si este valor es igual a 0, *bioNMF* aplica el algoritmo de NMF clásico, y cuanto más cercano a 1, más rala será la representación de los datos.
Fuente: [84].

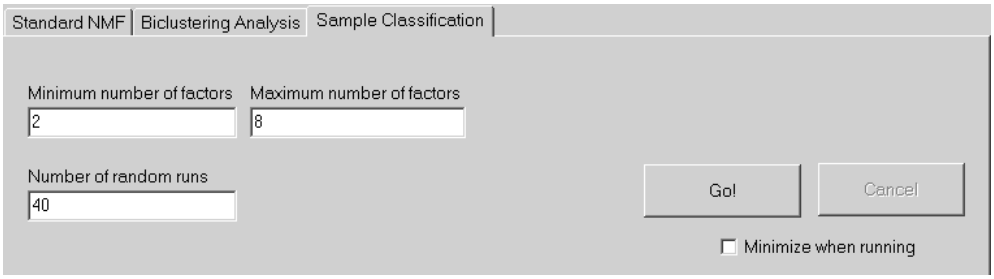

Figura 3.4: Módulo de clasificación de muestras

Los parámetros requeridos son i) *Rango de la factorización*: Selección del rango de valores de k que serán usados, desde el valor mínimo hasta el máximo. ii) *Number of random runs*: Número de ejecuciones del algoritmo de NMF con distintas condiciones iniciales.
Fuente: [84].

3.2. Implementación y requisitos

bioNMF fue implementado como una aplicación *stand-alone* para el sistema operativo *Microsoft Windows* utilizando *Borland Delphi*, versión 7 [93] (lenguaje `Object Pascal`). El análisis en *bioNMF* puede ser ejecutado en tres pasos consecutivos:

1. Selección del conjunto de datos a analizar.

2. Transformación de los datos para acomodarlos al requerimiento de no negatividad (sólo si es necesario).

3. Ejecución del análisis. Los tres principales tipos de análisis, explicados en la sección anterior, están incluidos en tres módulos independientes.

El ejecutable de la aplicación es de distribución gratuita y no tiene ningún requerimiento de autenticación. El programa, así como una descripción detallada de su uso, puede obtenerse en la página web http://www.dacya.ucm.es/apascual/bioNMF. El código fuente,

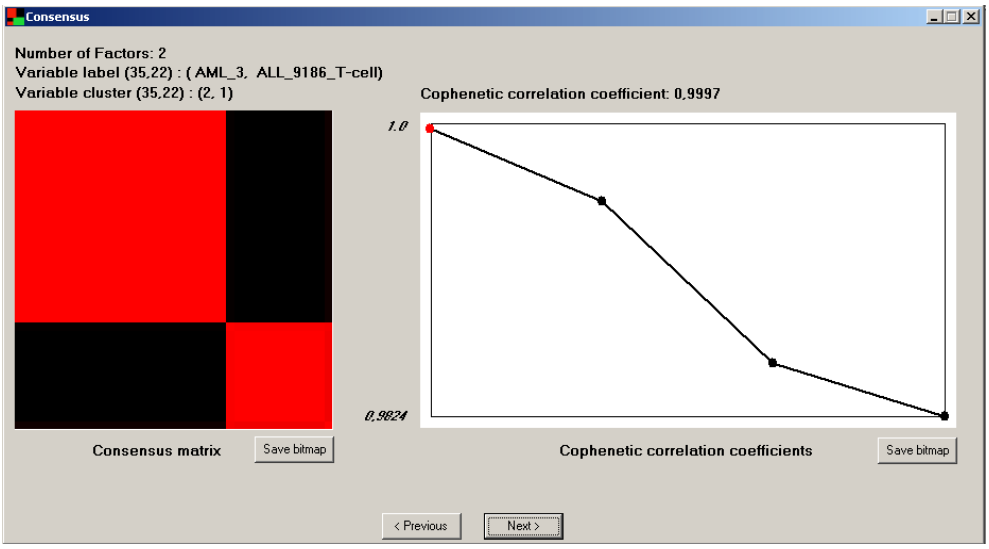

Figura 3.5: Ejemplo de clasificación de muestras

Este panel muestra la matriz consenso reordenada (panel izquierdo) y los valores del CCC (panel derecho) calculados para cada valor de k. La figura muestra la matriz consenso ordenada para un valor de $k = 2$ obtenida a partir de cincuenta ejecuciones del algoritmo. El gráfico de la derecha representa los valores del CCC obtenidos para un rango de $k = [2 \ldots 5]$.
Fuente: [84].

de libre distribución, está disponible en http://www.bioinformatics.org/bionmf/.

Sus únicos requerimientos son el sistema operativo, ya que sólo está disponible para *Microsoft Windows* (98, Me, 2000 o XP), y una resolución mínima de pantalla de 1024×768 para una mejor visualización.

Capítulo 4

Análisis de los algoritmos

En este capítulo se examinan los algoritmos de *Clasificación de muestras*, *Agrupamiento Doble* y de la factorización NMF, junto con sus posibilidades de optimización.

Como se mencionó anteriormente, la aplicación *bioNMF* está íntegramente implementada con el entorno de desarrollo `Borland Delphi` versión 7 [93] (en lenguaje `Object Pascal`) y específicamente para el sistema operativo *Windows* de Microsoft. Aunque existen compiladores de Delphi para Linux, no es viable reutilizar su código en `Pascal`, pues la mayoría de las tecnologías arquitectónicas y/o de programación actuales no tienen soporte para este lenguaje.

Por tanto, aunque esta aplicación se mantendrá como referencia *a nivel de funcionalidad*, en lo concerniente a los algoritmos se acudirá a las implementaciones originales, en `Matlab`, de sus respectivos autores [34, 35]. Ello se debe a que el proceso de paralelización se realizó de manera progresiva, enfocándose primero a aquellas secciones con mayor carga computacional. Por tanto, en las siguientes secciones se analizará el rendimiento de estos métodos en base a dichas implementaciones.

4.1. Análisis del algoritmo *Clasificación de muestras*

Como se describió en la sección 1.4.2, este método de *Análisis Exploratorio de Datos* distribuye las muestras o condiciones experimentales (columnas de la matriz de entrada) en diferentes clases, agrupando aquellas que tienen perfiles de expresión similares. Para ello se basa en el cálculo del *Coeficiente de Correlación Cofenética* (*CCC*), con el que se determina el *rango de factorización* (es decir, el número de clases) más adecuado a los datos.

Examinando el código en **Matlab** [92] de los autores del método [34] —cuyo esquema (pseudocódigo) se ilustra en la figura 4.1— se observa que este proceso es llevado a cabo por dos rutinas en sendas fases:

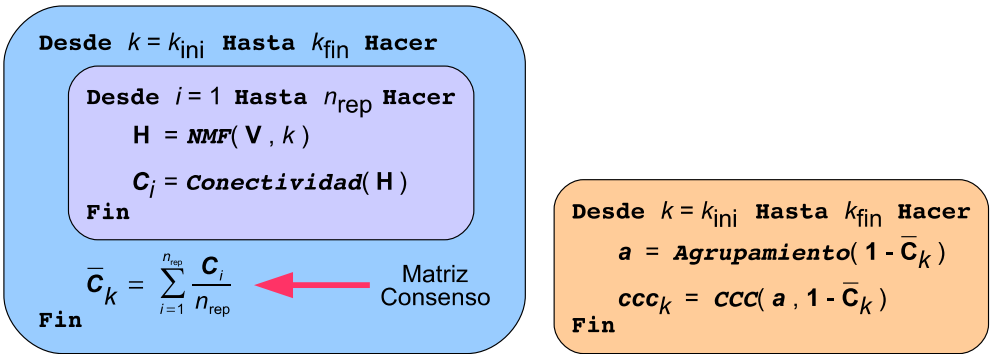

Figura 4.1: Implementación original de *Clasificación de muestras*

En la primera parte (izq.) se calcula la matriz Consenso ($\overline{\mathbf{C}}_k$) de cada rango de factorización candidato ($k \in [k_{\text{ini}}, \ldots, k_{\text{fin}}]$). Nótese que cada paso implica n_{rep} ejecuciones de NMF. Posteriormente, en la segunda parte (der.), se determinan todos los coeficientes CCC. El rango de factorización *ganador* será aquel donde el CCC obtenga un valor pico.
Fuente: [34].

- **nmfconsensus()**: recibe como parámetros de entrada a la matriz de datos ($\mathbf{V}$), los valores extremos de un conjunto de *rangos de factorización* candidatos (k_{ini} y k_{fin}) y el número de condiciones iniciales distintas a probar en cada uno de ellos (n_{rep}). El resultado es un vector de matrices Consenso $\left\{\overline{\mathbf{C}_1}, \ldots, \overline{\mathbf{C}_{k_{\text{fin}}}}\right\}$.
- **nmforderconsensus()**: recibe como parámetro la lista de matrices Consenso obtenidas en la etapa anterior y aplica un agrupamiento jerárquico a la matriz opuesta de cada una de ellas. El resultado es un vector de Coeficientes de Correlación Cofenética (uno por cada valor de k), así como otra lista de matrices Consenso. En esta nueva lista, cada matriz contiene los experimentos agrupados en clases, según el correspondiente valor de k.

Aunque ya se adelantó en la introducción y en la exposición de motivos, se intuye que la ***sección crítica*** (aquella que ocupa la mayor parte del tiempo de cómputo) corresponde a la primera etapa, particularmente al algoritmo de NMF.

También se evidencia que una de las mayores fuentes de consumo de memoria es el *vector de matrices Consenso*, representado como un *cubo* de dimensiones $k_{\text{fin}} \times m \times m$, siendo m el número de *columnas* de la matriz de entrada. Nótese que la primera dimensión (longitud del vector) *ignora* completamente el parámetro k_{ini}, por lo que no se llega a acceder a parte de la memoria reservada. Además, dado que todas estas matrices son *simétricas*, únicamente es necesario acceder a una de sus *diagonales*. Esta gestión

de memoria tan poco eficiente puede llegar a representar un desperdicio excesivo de los recursos del sistema, especialmente en matrices de datos con un gran número de columnas (parámetro m) o si se emplea un valor alto del parámetro k_{fin}.

La figura 4.2 ilustra esta situación. Se trata de la misma prueba mostrada en la exposición de motivación y objetivos (figura 2.1), pero desglosando respecto del total, tanto el tiempo empleado en la primera fase del proceso, como la memoria consumida por el vector de matrices Consenso.

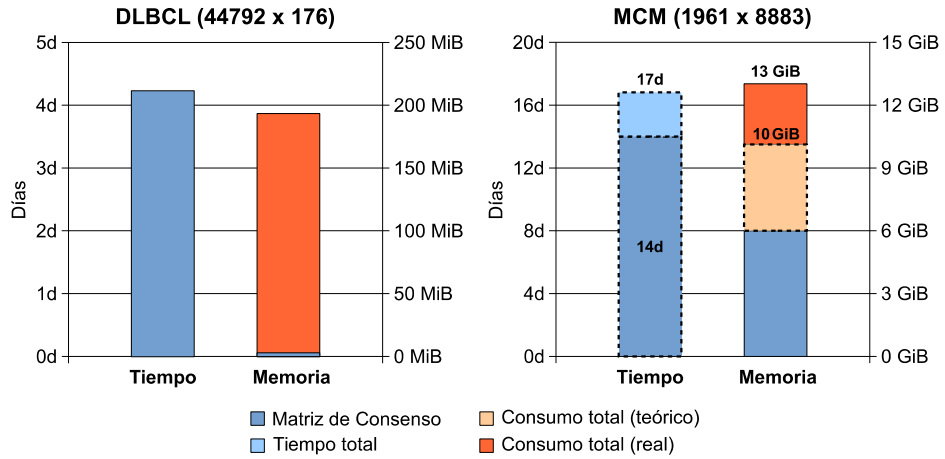

Figura 4.2:
Desglose del rendimiento de *Clasificación de muestras* (implementación original [34]).

Figura similar a la 2.1, pero destacando respecto del total, tanto el tiempo empleado en las 360 ejecuciones de NMF, como la memoria consumida por el vector de matrices Consenso.

En el conjunto de datos *DLBCL*, con pocas columnas, todo el tiempo computo corresponde a las ejecuciones de NMF (la segunda fase solo duró unos pocos segundos) y las matrices Consenso ocupan muy poca memoria.
Por su parte, *MCM* posee un gran número de columnas, por lo que las matrices Consenso ocupan el 60 % de la memoria consumida; al menos del total teórico, pues a nivel empírico existe una diferencia causada por la (deficiente?) gestión de memoria de `Matlab`. Igualmente se hace notable el tiempo empleado en la segunda fase (cuatro días).

Otro dato importante que también se desprende de este análisis es la completa ***independencia*** de todas las ejecuciones de NMF, por lo que sería posible ejecutarlas de manera simultánea. No obstante, se trata de una paralelización de grano bastante grueso que no resuelve la complejidad inherente al algoritmo de NMF, por lo que su aplicación sería adicional al proceso de optimización de este último.

4.2. Análisis del algoritmo *Agrupamiento doble*

Como se explicó en la sección 1.4.1, este algoritmo es empleado para el agrupamiento simultáneo de genes y experimentos en matrices de expresión génica. Un esquema muy básico de este algoritmo puede verse en la figura 4.3.

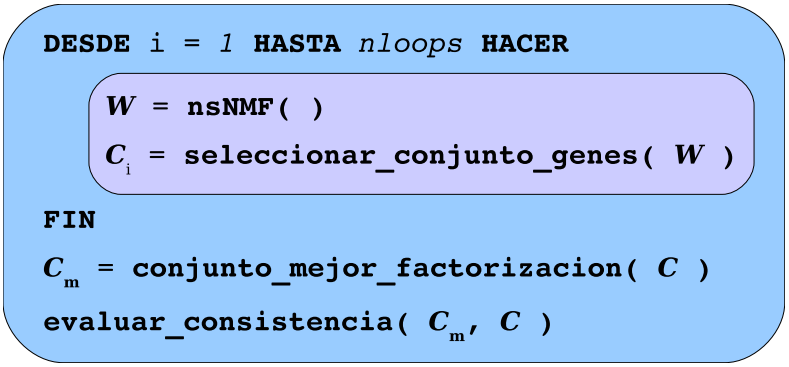

Figura 4.3: Esquema básico del método *Agrupamiento doble de datos*

Se ejecuta NMF *nloops* veces seleccionando en cada momento un conjunto representativo de genes (módulos). A continuación, se evalúa la consistencia de los módulos correspondientes a la mejor de las *nloops* factorizaciones en el conjunto de módulos seleccionados en el resto de factorizaciones. Un módulo se considera consistente si su tasa de aparición en el conjunto supera un cierto umbral.

La implementación de este algoritmo guarda cierto parecido con el de *Clasificación de ejemplares* en el sentido de ejecutar NMF un cierto número de veces (*nloops*), siendo estas ejecuciones independientes entre sí. Por tanto, se podrá aplicar el mismo proceso que el anterior.

4.3. Análisis de la factorización NMF

Como se aprecia en el pseudocódigo 1.1 de la página 18, la factorización *NMF* está compuesta por diversas operaciones de álgebra lineal tales como productos de matrices, reducciones a vector y otras operaciones punto a punto. Todas ellas exponen un alto grado de paralelismo ***a nivel de datos*** debido que están compuestas por operaciones aritméticas de *bajo nivel* (sumas y productos entre escalares) que se ejecutan con operandos distintos y mayoritariamente independientes.
Por el contrario, el *flujo del programa* es totalmente ***secuencial***.

De manera simular a otros algoritmos de álgebra lineal, el tamaño de todas las estructuras de datos guardan una relación directamente proporcional a las dimensiones del conjunto de datos de entrada. Por tanto es posible aplicar un modelo de *partición de datos* sobre

el número de filas y de columnas de las tres matrices, **V**, **W** y **H**. La figura 4.4 muestra el resultado de este proceso al aplicarlo a la regla de actualización de la matriz **H**.

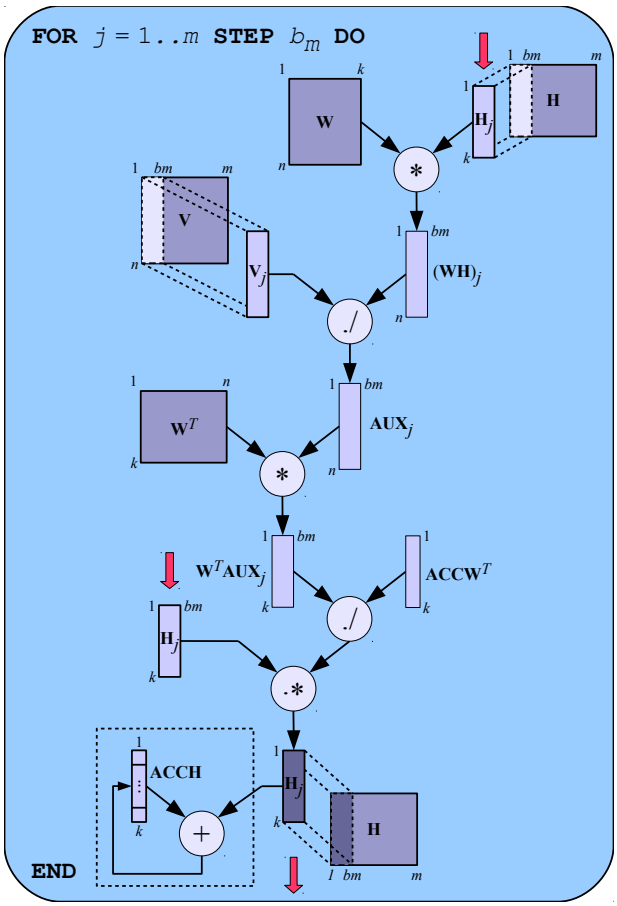

Figura 4.4: Esquema de partición de datos del algoritmo de NMF (matriz H)

Actualización de la matriz **H** en bloques de b_m columnas. Esta operación requiere del correspondiente conjunto de columnas de **V** y de toda la matriz **W**. Los símbolos "$.*$" y "$./$" denotan operaciones punto a punto entre matrices.

Por motivos de simplificación, en la partición no se toma en cuenta el *rango de factorización* (parámetro k) que también influye en el tamaño de las estructuras; de hacerlo, sería necesario un segundo nivel de partición datos. Como se observa, para actualizar un conjunto de columnas de la matriz **H** es necesario contar con el mismo conjunto de columnas de **V** y con toda la matriz **W**. Nótese que la zona recuadrada, en la parte inferior izquierda de la figura, representa la operación de reducción a vector que es necesaria en la regla de actualización de la matriz **W**.
Aplicando este modelo a **W** se obtiene un esquema análogo. En este caso, para actualizar un conjunto de filas de esta matriz se requiere del correspondiente conjunto de filas de **V** y de toda la matriz **H**.

La gran ventaja de este esquema es la posibilidad de ajustar el tamaño de los bloques

(parámetro b_m de la figura 4.4) y con ello, el ***grado de paralelismo*** expuesto. En efecto, disponiendo de la cantidad suficiente de unidades de cómputo sería posible actualizar simultáneamente todas las columnas de $\mathbf{H}$ (es decir, estableciendo $b_m = 1$), y luego proceder de manera análoga con las filas de $\mathbf{W}$. En caso contrario, estos bloques se pueden agrupar en conjuntos más grandes ($b_m > 1$) y ser procesados de manera iterativa. Obviamente en cualquiera de estas situaciones habrá que tener en cuenta el coste de sincronización de las porciones actualizadas.

Finalmente, en este proceso de análisis se aprecia ***otro nivel de paralelismo***, de grano más fino, correspondiente a las ***operaciones aritméticas*** que conforman cada *operador* de álgebra lineal. Así, en las operaciones *entre vectores* esto implica procesar cada valor escalar de manera *independiente*. Ejemplo de ello es la segunda operación de la figura 4.4 (división punto a punto).

4.4. Modelo de optimización

En el análisis practicado a estos tres algoritmos se han identificado distintos tipos de operaciones que exponen diversos grados y niveles de paralelismo. Dado que será necesario entonces variar las estrategias de optimización a aplicar en cada uno de ellos, se propone un modelo de optimización, basado en capas, que permite la aplicación de tales estrategias de manera selectiva o combinada. El modelo es el siguiente:

- El ***nivel inferior*** abarca a todas las operaciones aritméticas, entre escalares, que componen *cada* "operador" de álgebra lineal (por ejemplo, un producto de matrices). En este caso, la estrategia de optimización consiste en aprovechar el paralelismo a nivel de *datos* que se expone.
- El ***nivel intermedio*** comprende el *conjunto* de operaciones de álgebra lineal que conforman el algoritmo de NMF. La estrategia en este nivel también consiste en aprovechar el paralelismo a nivel de *datos* que se expone. Al respecto, es posible variar el *grado de partición de datos*.
- El ***nivel superior*** corresponde al paralelismo a nivel de *cómputo* que representan a todas las instancias de NMF que son ejecutadas para determinar el mejor rango de factorización.

Capítulo 5

Proceso de optimización

En el capítulo anterior se determinó el modelo de paralelismo a aplicar para la optimización de los algoritmos. Dicho modelo está basado en capas, en función de los niveles de granularidad de sus operaciones. A continuación, se describen las diversas tecnologías de altas prestaciones que se han empleado para aprovechar el paralelismo de cada nivel.

5.1. Paralelismo de grano fino

Este nivel corresponde a los productos de matrices y otras operaciones algebraicas de las que se compone NMF. Se trata por tanto de la sección más crítica de todo el proceso, de manera que cualquier mejora que se realice tendría un impacto muy importante en el rendimiento general.

5.1.1. Optimización mediante bibliotecas de funciones de álgebra lineal

Un primer intento consistió en emplear una biblioteca de rutinas optimizadas de álgebra lineal, denominada ***ATLAS*** (***Automatically Tuned Linear Algebra Software***) [94]. De manera análoga a otras bibliotecas similares, tales como *MKL* (*Intel Math Kernel Library*) [95] y *OpenBLAS* [96], ATLAS ofrece tres niveles de operaciones siguiendo la funcionalidad especificada por el estándar (*de facto*) ***BLAS*** (*Basic Linear Algebra Subprograms*) [97]:

- **Nivel 1:** Operaciones que involucran a **vectores y escalares**, tales como producto escalar, norma vectorial, reducción de vector a escalar mediante sumas, posición del valor máximo, rotación en el plano, copia de un vector a otro, suma y escalado de vectores, etc.
- **Nivel 2:** Operaciones entre **vectores y matrices**, tales como productos matriz-vector, actualizaciones de rangos 1 y 2, y resolución de sistemas triangulares lineales.
- **Nivel 3:** Operaciones entre **matrices**, tales como producto de matrices, actualizaciones de rango k y resolución de sistemas triangulares lineales múltiples.

En estos tres niveles se distinguen casos para matrices y vectores compuestos por números *reales* o *complejos*, tanto de *precisión simple* como *doble*. Por su parte, en los niveles 2 y 3 existen operaciones específicas para matrices *simétricas* o *hermitianas* y *triangulares*, que pueden, además, estar almacenadas en memoria en un formato empaquetado.

ATLAS también implementa un pequeño conjunto de las operaciones que proporciona otra biblioteca, ***LAPACK*** (***Linear Algebra PACKage***) [**?**], que extiende la funcionalidad de *BLAS*. Entre estas funciones se encuentra la resolución de sistemas generales de ecuaciones lineales, factorización LU y la inversa de una matriz para distintas versiones de matrices y diferentes tipos de datos [94].

Actualmente se utiliza como base de cálculo en diversos entornos de programación matemática de alto nivel, tales como `Matlab` [92], `Octave`, `Maple` o `Mathematica`.

A continuación se detalla el funcionamiento de esta biblioteca y posteriormente se describe su aplicación en la optimización de NMF.

5.1.1.1. Funcionamiento de la biblioteca *ATLAS*

La particularidad de la biblioteca *ATLAS* consiste en su capacidad para ***ajustarse automáticamente*** a la configuración específica del sistema donde se instala (de ahí su nombre), lo que le concede un enorme grado de portabilidad y flexibilidad frente a nuevas arquitecturas. Para ello, los autores se basan en un paradigma que han denominado ***AEOS*** (***Automated Empirical Optimization of Software***) [94] consistente en un conjunto de pruebas que se realizan al momento de la instalación. En ellas se ensayan diversas técnicas de optimización de código comprobando su efectividad en el sistema

destino. En función del resultado se genera y/o se adapta el código que conformará finalmente las rutinas de la biblioteca.

Entre los diversos parámetros que se determinan durante las pruebas de instalación figura el tamaño de la memoria cache de nivel 1 (tanto para instrucciones como para datos) y, de manera implícita, de nivel 2; el tamaño mínimo de una matriz en la que se puede asumir la sobrecarga de realizar ciertas copias locales; tamaño de las submatrices con las que se puede operar sin producir fallos de cache (factor de bloqueo); cantidad de registros disponibles; etc.

En base a esta información es posible tomar ciertas decisiones —bien sea en tiempo de compilación o de ejecución— sobre el comportamiento de los algoritmos de las rutinas. Por ejemplo, el tamaño de desenrollado de los bucles anidados y el orden de ejecución de los mismos, el patrón de acceso a memoria en función de las submatrices alojadas en la cache, el reordenado de instrucciones cuando el compilador y/o la arquitectura no posee dicha capacidad, etc.

La aplicación de todas estas técnicas de optimización depende del nivel de operaciones BLAS. Así, en las rutinas del **nivel 1** únicamente se abarca el orden de las instrucciones y algunas optimizaciones sencillas en los bucles. Dado que son rutinas muy simples, representarán una (pequeña) mejora solo cuando no se disponga de un compilador moderno capaz de aplicar estas técnicas de manera automática.

Por su parte, en las operaciones del **nivel 2** ya es posible aplicar bloqueo de memoria y de registros, lo que permite reutilizar algunos operandos en los vectores y reducir la complejidad de acceso a memoria, de O(N^2) a O(N); no obstante, en el caso de las matrices se mantiene en O(N^2). Por tanto, la ganancia que se puede obtener, si bien sería modesta, es superior a la del nivel 1 debido a que los bucles son más complejos y menos compiladores son capaces de optimizarlos.

Finalmente, en el **nivel 3**, es posible aplicar todas las técnicas descritas permitiendo reducir la complejidad de acceso a memoria, de O(N^3) a aproximadamente O(N^2). Por tanto, la ganancia total puede ser de órdenes de magnitud ya que los tres bucles anidados del producto de matrices suelen ser bastante complejos como para que un compilador pueda optimizarlos automáticamente.

5.1.1.2. La factorización NMF mediante la biblioteca *ATLAS*

La implementación de esta versión del algoritmo de NMF es bastante sencilla y rápida, pues en los lugares donde se requiere ejecutar un producto de matrices, basta con

insertar una llamada a la función correspondiente y con los parámetros adecuados. Dado que las matrices son densas y pueden tener un tamaño arbitrario, en todos los casos se ha empleado la versión general de esta función (`gemm`).

No obstante, el resto de operaciones que componen la factorización NMF se han implementado manualmente, excepto donde ha sido posible emplear alguna función del nivel 1 de BLAS, como en el caso de las reducciones de matriz a vector.

5.1.1.3. Pruebas y resultados en ATLAS

Al comparar el rendimiento de esta versión con el de una implementación sencilla en `Matlab` se obtienen los resultados mostrados en la figura 5.1. En ella se aprecia una ligera ganancia en el tiempo de cómputo (*speedup*) que oscila entre el 21 % y el 54 %. La plataforma de prueba es *antigua* debido a que estos resultados se utilizarán, más adelante, como *referencia* en las pruebas de rendimiento de las primeras GPU empleadas. Tales dispositivos son de hace casi una década, por lo que no sería justo compararlos con procesadores (CPU) de última generación.

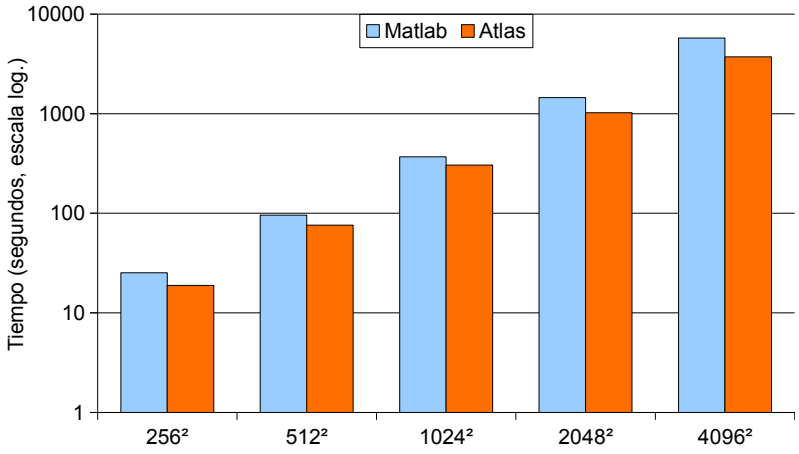

Figura 5.1: NMF en ATLAS

Tiempos de ejecución del algoritmo de NMF (divergencia) implementado con ATLAS para matrices sintéticas de distintas dimensiones. Pruebas realizadas con $k = 64$ y 2000 iteraciones (sin evaluar convergencia ni condiciones de parada).
Plataforma de prueba: *Intel Pentium 4 Prescott* (3,4 GHz, 2 GiB RAM, 800 MHz FSB, 2 MB cache L2), `ATLAS 3.6.0` y `Matlab 7.3.0.298 (R2006b)`.

Si bien la figura 5.1 corresponde sólo a la variante *Divergencia* (ecuaciones (1.4) en la sección 1.3.1), los tiempos del resto de versiones de NMF son bastante parecidos, tanto en la ganancia de tiempo respecto a `Matlab` como en términos absolutos. Prueba de ello son los tiempos expuestos en la gráfica 5.2 para una matriz de dimensiones

concretas. Por tanto, a partir de ahora, únicamente se mostrarán los tiempos de cómputo referentes a dicha variante.

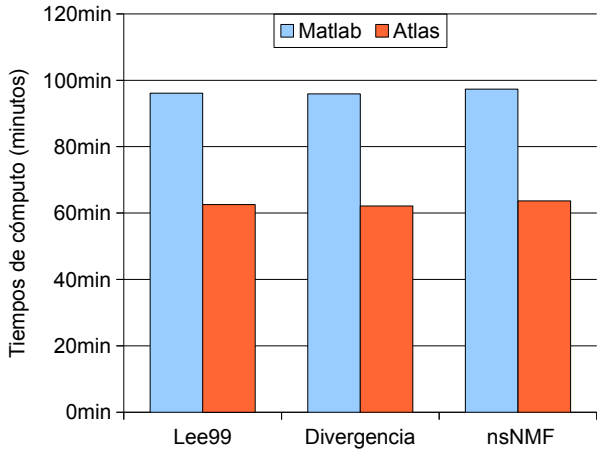

Figura 5.2: NMF en ATLAS

Comparación en los tiempos de ejecución de diversos algoritmos de NMF implementados con ATLAS y con `Matlab`, para una matriz de dimensiones 4096×4096. Pruebas realizadas con $k = 64$ y 2000 iteraciones (sin evaluar convergencia ni condiciones de parada). El tiempo aparece medido en horas y minutos.

Aunque el empleo de la biblioteca ATLAS supuso una leve mejora respecto de `Matlab`, esta solución se basa en tan solo unas pocas unidades de cómputo (*cores*), por lo que no aprovecha todo el paralelismo a nivel de datos existente.

5.1.2. Optimización mediante Unidades de Procesamiento de Gráficos (GPU)

Debido a las limitaciones de la versión anterior de NMF, se decidió aprovechar las capacidades de cómputo de una tecnología en pleno auge: las ***Unidades de Procesamiento de Gráficos*** (***GPU***, del inglés *Graphics Processing Unit*).

Para representar un objeto 3D en una pantalla bidimensional es necesario calcular su proyección sobre un plano, labor que requiere diversas operaciones trigonométricas y de álgebra lineal. Inicialmente, estas acciones eran llevadas a cabo en el procesador principal del sistema (***CPU***, del inglés ***Central Processing Unit***), por lo que las ***tarjetas gráficas*** presentes en computadores personales (PC) y otros equipos informáticos, únicamente se ocupaban de representar la imagen 2D final. Sin embargo, ante el enorme e incesante crecimiento del mercado de videojuegos, en el que los objetos 3D son cada vez más complejos y detallados, todas estas operaciones se han ido trasladando fuera de

la CPU, conllevando a la aparición y desarrollo de las *Unidades de Procesamiento de Gráficos* (*GPU*).

Teniendo en cuenta el tipo y, sobre todo, la *intensidad* de las operaciones de cálculo que deben realizar estos dispositivos, su arquitectura se ha constituido de una manera distinta a la de los procesadores tradicionales. Ello se evidencia en la figura **??**, donde se realiza una comparación esquemática de ambos modelos arquitectónicos. Mientras que en las CPU clásicas predominan las estructuras de memoria cache y de control de flujo de programas, en los procesadores de gráficos se dedica la mayor parte de la circuitería (es decir, de los transistores) a las unidades de cómputo [98]. En el apéndice A se describe de manera un poco más detallada este modelo arquitectónico y su evolución, así como el proceso llevado a cabo para la representación de gráficos en pantalla.

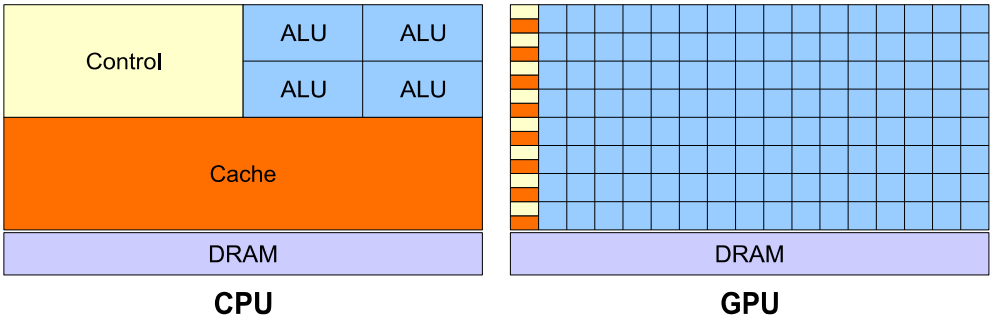

Figura 5.3: Comparativa de las arquitecturas de CPU y GPU (fuente: [98])

En los procesadores de gráficos (GPU), las unidades funcionales (*ALU*, del inglés *Arithmetic Logic Unit*) son el elemento central del diseño, relegando las estructuras de control y almacenamiento (eje fundamental de los procesadores superescalares clásicos) a un segundo plano.

Gracias a esta *especialización* en su arquitectura, los procesadores de gráficos han alcanzado un rendimiento muy superior al de las CPU tradicionales en ese tipo de tareas. Un ejemplo de ello aparece en la figura 5.4, donde se ilustra la evolución en la potencia de cálculo de las GPU de NVIDIA.

Además de su arquitectura, también han ido evolucionando las herramientas para su programación: inicialmente utilizando lenguajes de dibujado de gráficos como `OpenGL` [99] y `Cg` [100], y posteriormente aprovechando entornos más genéricos como ***CUDA*** (*Compute Unified Device Architecture*) [98] o el más reciente, `OpenCL` [101]. Esto ha permitido emplear los dispositivos GPU como ***coprocesadores*** genéricos (o de propósito general) que se encarguen de ejecutar aquellos programas, o partes de algoritmos, que contengan operaciones de álgebra lineal. Además, dado que ***ya están presentes*** en una amplia variedad de sistemas informáticos de uso común (PC de sobremesa, portátiles, sistemas empotrados como teléfonos móviles, tabletas, consolas de

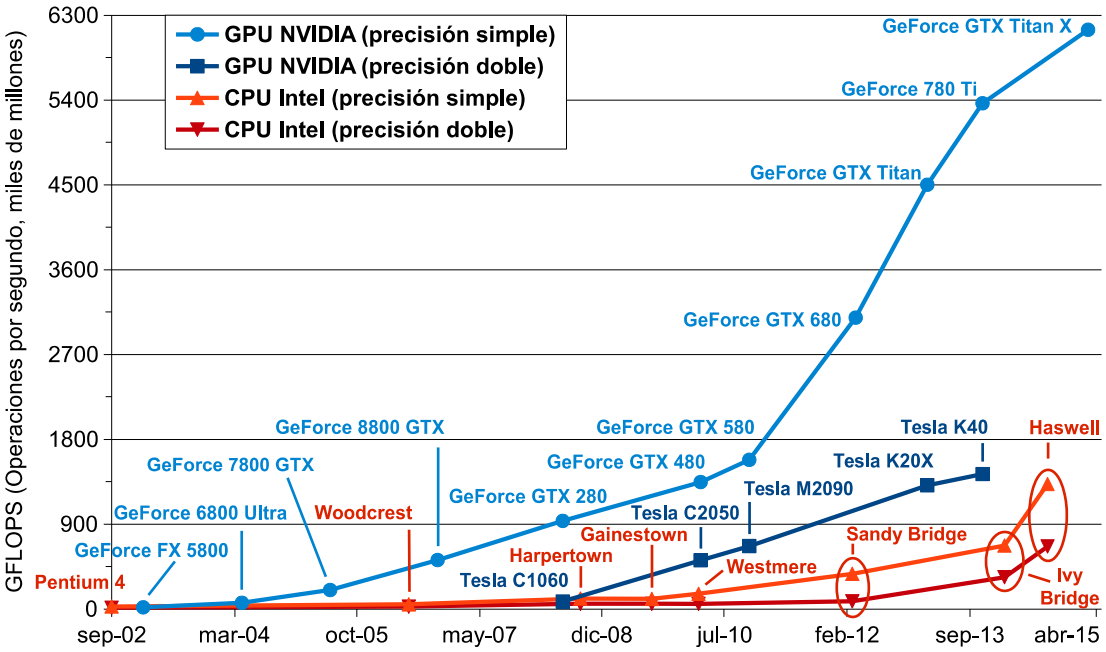

Figura 5.4: Evolución de los procesadores de gráficos de NVIDIA (fuente: [98])

Rendimiento de varios modelos de GPU de NVIDIA y de diversos procesadores convencionales de Intel, medido en miles de millones de *operaciones de punto flotante por segundo* (***GFLOPS***). Se observa que los dispositivos GPU pueden ejecutar muchas más operaciones por segundo que las CPU.

videojuegos, etc), es posible emplearlos como ***equipos de alto rendimiento "de bajo coste"***, frente a los sistemas multiprocesadores tradicionales (*clusters*).

Tales características han llevado a la generalización de su uso por parte de la comunidad científica, dando lugar además a la aparición del ***Cómputo de Propósito General en Procesadores de Gráficos*** (*General-Purpose computation on GPU* o ***GPGPU***) [102–104] como campo de investigación [105, 106].

Entre las muchas áreas de aplicación de los procesadores de gráficos se encuentra el procesamiento de imágenes [107], algoritmos de ordenamiento [108], química cuántica [109] y ciencias físicas [110]. En Bioinformática y Biología Computacional se han empleado para el análisis de datos de expresión génica [111], alineamiento de secuencias [112], acoplamiento de proteínas [113], simulación de sistemas celulares [114], entre otros [115]. También han surgido diversas propuestas sobre operaciones específicas de álgebra lineal (por ejemplo, productos de matrices [116] o factorización LU [117]), así como bibliotecas genéricas de funciones [118].

Con respecto a la factorización NMF, ya existen algunas implementaciones en GPU [43, 47, 88, 89] destinas a algunas áreas específicas de investigación. Sin embargo, ninguna de ellas toma en cuenta la ***capacidad de los recursos de almacenamiento*** disponibles

con respecto al tamaño de los datos.
En efecto, muchas arquitecturas GPU son presentadas al mercado en forma de ***dispositivos individuales***, que se comunican con el resto del sistema mediante un bus de datos (por ejemplo, *PCI Express*) y están dotados de una memoria de uso exclusivo cuya capacidad suele ser ***mucho menor*** que la de la memoria principal. Por tanto, en el caso de matrices de gran tamaño, que superen la capacidad de la memoria del dispositivo GPU, es necesario un ***procesamiento iterativo por bloques*** que incluya transferencias entre la memoria CPU (principal) y la del procesador de gráficos. La repercusión en el rendimiento de este tipo de operaciones suele ser *muy significativa*, por lo que debe hacerse de manera cuidadosa.

A continuación se resume brevemente el ***modelo de programación*** utilizado para adaptar algoritmos a la arquitectura GPU. Posteriormente, se expondrán dos propuestas de implementación de la factorización NMF, en las que se toma en cuenta la *restricción de memoria* antes mencionada. En la primera versión se emplean los lenguajes de síntesis de gráficos `OpenGL` y `Cg`, mientras que la segunda se utiliza el paradigma actual de NVIDIA, *CUDA*.

5.1.2.1. Modelo de programación: el Procesamiento de Flujos (*Stream Processing*)

Ante el desarrollo de los procesadores gráficos y la posibilidad de ejecutar ***funciones programadas por el usuario***, la comunidad científica comenzó a interesarse en estos dispositivos para la ejecución genérica algoritmos de álgebra lineal. Por ello, se comenzó a investigar en modelos de programación que permitiesen adaptar los algoritmos a esta arquitectura. Uno de ellos es el ***Procesamiento de flujos*** (*Stream Processing*) en el que los datos se agrupan en ***flujos*** (*streams*) y son procesados por ***núcleos*** (*kernels*) que generan nuevos *flujos* de salida.

Este paradigma no es nuevo y podría considerarse como una actualización del *procesamiento vectorial* o de las antiguas arquitecturas *data flow*. Tampoco es exclusivo de los procesadores de gráficos, pues existen diversas arquitecturas (Imagine [119], Raw [120] o Cell [121]) especializadas en este modelo, así como propuestas que intentan utilizar este paradigma en procesadores de cómputo general [122].

La revisión de este clásico se debió a la aparición de aplicaciones en campos como multimedia, el procesamiento de señales o el tratamiento de imágenes, que requieren ejecutar miles de millones de operaciones por segundo para satisfacer las restricciones de tiempo real. Afortunadamente, la mayoría de estas aplicaciones exhiben una enorme

cantidad de *paralelismo*, especialmente a ***nivel de datos***. Esta característica permite modelarlas fácilmente de acuerdo con este nuevo paradigma: flujos de datos procesados de forma ***independiente*** por pequeños núcleos de cómputo extremadamente eficientes. Un ejemplo de aplicación al campo multimedia se muestra en la figura 5.5.

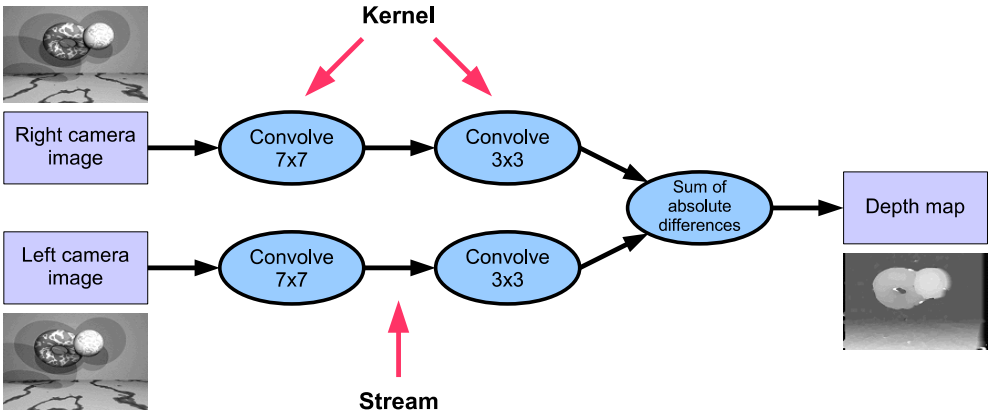

Figura 5.5: Procesamiento de flujos. Ejemplo de aplicación multimedia (fuente: [123]).

Los óvalos representan las operaciones (*kernels*) que se ejecutan sobre uno o varios flujos de datos (*streams*) que recibe, generando otro flujo de salida.

Una de las ventajas del procesamiento de flujos radica en que las operaciones de cómputo quedan *desacopladas* de los accesos a memoria, ofreciendo así más oportunidades de ocultar latencias y permitiendo una escalabilidad *transparente* al programador. Esto es, para mejorar el rendimiento, *sin necesidad de modificar el código*, basta con añadir más unidades de cómputo de manera que se incremente la cantidad de *kernels* que se ejecutan simultáneamente.

En el caso de las arquitecturas GPU se comenzó utilizando el *procesador de fragmentos* como unidad de cómputo, ya que ofrecía mejores prestaciones que el de vértices (los detalles de la arquitectura GPU se describen en el apéndice A). Así, los *núcleos* se implementaban como ***programas de fragmentos***, mientras que los *flujos* de datos se representaban como ***texturas*** [106].

Inicialmente era necesario utilizar lenguajes de síntesis de gráficos como `OpenGL` [99] (y/o `Cg` [100] en el caso de los *kernels*). Ello obligaba a tener un conocimiento profundo del funcionamiento interno de las GPU, pues era necesario implementar los algoritmos *como si se tratasen* de aplicaciones gráficas.

Aunque más adelante se propusieron algunos lenguajes (como `Brook` [124]) y *frameworks* (como el empleado en [125]) que ahorraban la necesidad de aprender `OpenGL`, el modelo de programación de los dispositivos GPU estaba todavía muy orientado a las aplicaciones de gráficos, por lo que seguía existiendo un cierto grado de dificultad para

adaptar algunos algoritmos de álgebra lineal.

Afortunadamente todo ello cambió con la evolución de las arquitecturas GPU y el desarrollo del entorno de programación de NVIDIA, ***CUDA*** (*Compute Unified Device Architecture*) [98]. Aunque este paradigma también se basa en el modelo de *procesamiento de flujos*, se trata de una *generalización* del mismo, mucho más flexible, que permite realmente emplear las GPU como coprocesadores de propósito general.

5.1.2.2. Implementación en `OpenGL`

Al igual que otros algoritmos de álgebra lineal, la factorización NMF puede expresarse fácilmente bajo el paradigma de *procesamiento de flujos* asignando cada una de las operaciones a un *kernel* diferente, e interpretando las correspondientes matrices como *flujos de datos*. Cada *kernel* extraído de este modo recibiría dos flujos de entrada (dos matrices) y devolvería una nueva matriz resultado como flujo de salida. Sin embargo, en muchos problemas prácticos no suele ser factible establecer este tipo de correspondencia 1-a-1 entre *kernels* y operaciones matriciales.

En el caso concreto que nos ocupa, las dimensiones de las matrices de datos $\mathbf{V} \in \mathcal{M}_{n\times m}(\mathbb{R})$, $\mathbf{W} \in \mathcal{M}_{n\times k}(\mathbb{R})$ y $\mathbf{H} \in \mathcal{M}_{k\times m}(\mathbb{R})$ pueden llegar a exceder la capacidad de la memoria, haciendo inviable esta estrategia. Es imperativo por lo tanto, descomponer la descripción matricial del algoritmo en etapas cuyos flujos de entrada y de salida no superen las limitaciones impuestas por el sistema de memoria, y a partir de estas, extraer los correspondientes *kernels*.

En este sentido, las tres matrices serán procesadas en bloques de dimensiones $bln \times blm$, $k \times blm$ y $bln \times k$, respectivamente, donde $bln \leq n$ y $blm \leq m$. Los valores de bln y blm se establecerán en tiempo de ejecución en función de las dimensiones de $\mathbf{V}$ y de la memoria disponible. Este esquema se ilustra en la figura 5.6. Nótese que se trata de una ampliación del mostrado en la figura 4.4.

Básicamente, para actualizar una fila de \$**W** es necesario calcular una fila de **AUX2**, la cual se obtendría con el producto de matrices entre $\mathbf{AUX1}_i$ (fila i de **AUX1**) y $\mathbf{H}^t$, pero para ello sería necesario cargar en la memoria de la GPU toda la matriz **H** y una fila completa de **V**, y como se mencionó anteriormente, esto puede ser inviable debido a las restricciones de memoria.

Por tanto, la fila de **AUX2** se debe calcular, de forma iterativa, en un bucle anidado que procese **H** columna a columna y la fila i de **V** elemento a elemento, y acumulando el resultado en **AUX2**.

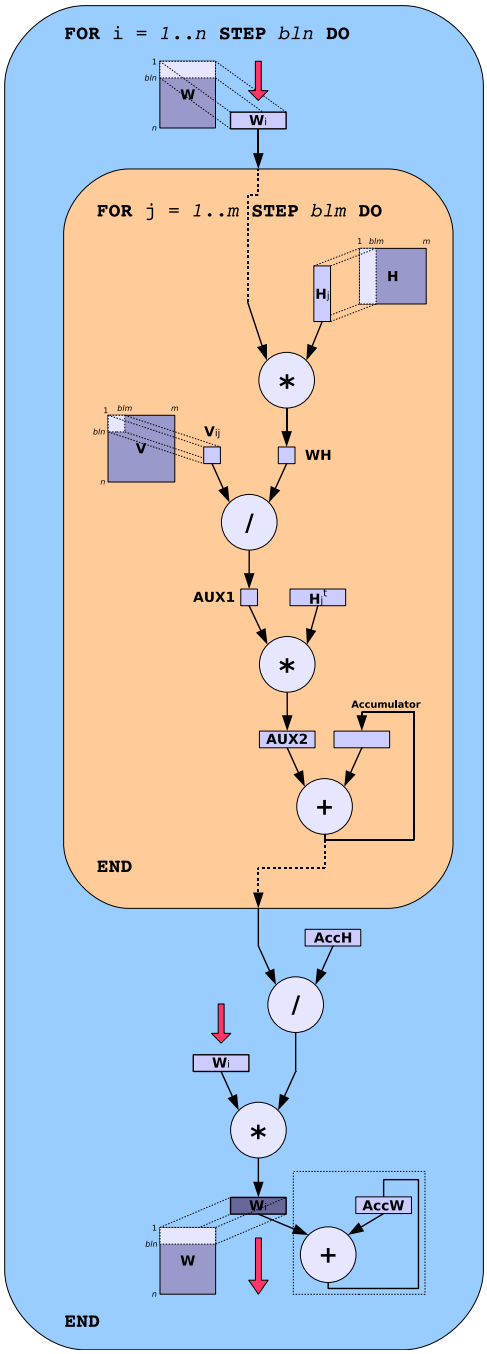

Figura 5.6: NMF en OpenGL

Actualización de la matriz **W** en bloques de *bln* filas.

El proceso de actualización de la matriz **H** se realiza de una forma similar, solo que en bloques de *blm* columnas.

Otros detalles de implementación La primera decisión de diseño específica que se debe realizar es precisamente cómo efectuar el *mapping* entre *streams* (bloques de matrices en nuestro problema) y texturas. Hay que tener en cuenta que cada elemento

de textura (*texel*) consta de cuatro componentes (canales RGBA) y que conviene utilizar todos ellos para aprovechar la capacidad de procesamiento SIMD de los procesadores de fragmentos.

Existen diferentes alternativas: asignar cuatro valores de una misma fila a cada texel, cuatro de una columna, o cualquier otra combinación. Dependiendo del *layout* escogido, puede favorecerse el procesamiento por filas o por columnas. Dado que para este algoritmo son necesarios ambos tipos de procesamientos, se ha optado por utilizar un *layout* 2×2, que permite un tratamiento más homogéneo de filas y columnas, y la reutilización de algunos de los *kernels*.
No obstante, cualquiera de estos *layouts* impone el empleo de datos de *relleno* (*padding*) para matrices o bloques con un número impar de filas y/o de columnas. Sin embargo, datos de relleno que no alteran el cómputo en un kernel, bien pueden modificar el resultado de otro (por ejemplo, el valor 0 no afecta a las sumas pero sí a los productos), lo que obliga a cambiar todos estos elementos entre ejecuciones de kernels distintos.

Otra posibilidad (que fue la adoptada), es la de implementar diversas versiones de un mismo kernel para procesar esas zonas particulares de las matrices. Es necesario hacer estas versiones como *kernels* independientes debido a que en las primeras arquitecturas GPU, no es posible, o es muy ineficiente, incluir en el código de un kernel estructuras condicionales que impliquen una instrucción de salto. Por ejemplo, en el caso de los bucles, estos son completamente desenrollados, imponiendo la necesidad de conocer en tiempo de compilación el número de iteraciones.

Otra característica importante es que al compilar un kernel, el número de instrucciones en código objeto no puede sobrepasar un cierto valor. Esto limita bastante el número de iteraciones que puede dar un bucle, teniendo que compilar diversas versiones del mismo kernel, con distintas iteraciones, para alcanzar el número deseado. Este fue el caso del kernel para productos de matrices, donde el producto escalar entre vectores fila y/o columna está implementado mediante bucles. Dado que para esta operación no se conoce en tiempo de compilación la longitud de esos vectores, fue necesario compilar diversas versiones del mismo kernel, cada una con un número de iteraciones potencia de dos, para que en tiempo de ejecución se seleccionen aquellas que permitan aproximar este valor.

Finalmente, es de resaltar que para optimizar el rendimiento no se ha hecho uso de *multitexturing*, es decir, todos los *streams* se han emplazado en una única textura, y se ha utilizado un gestor de memoria dinámica específico que intenta minimizar la fragmentación que introduce una organización de memoria 2D.

5.1.2.3. Pruebas y resultados en OpenGL

A continuación, se presentan los resultados de la implementación de NMF en la GPU cuyas características se indican en la Tabla 5.1.

Cuadro 5.1: Características de la GPU 7800GTX

Marca y modelo	*NVIDIA GeForce 7800GTX*
Año	2005
Arquitectura	G70
Bus	*PCI Express*
Memoria de vídeo	256 MiB
Reloj interno	430 MHz
Reloj memoria	1.2 GHz GDDR3
Interfaz memoria	256-bitshape
Ancho banda	38.4 GiB/s
Número de procesadores de fragmentos	24
Relleno texturas	10320 MTexels/s

En la figura 5.7 se muestran los tiempos de cómputo en comparación con los de las implementaciones en ATLAS y en `Matlab` (expuestos en la figura 5.1).

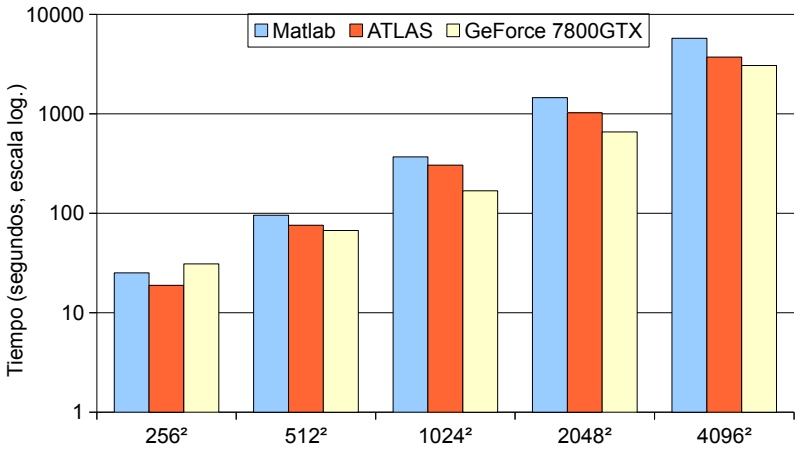

Figura 5.7: NMF-OpenGL

Tiempos de ejecución del algoritmo de NMF implementado con `Matlab`, ATLAS y la GPU 7800GTX de NVIDIA, para matrices de distintas dimensiones. Pruebas realizadas con $k = 64$ y 2000 iteraciones (sin evaluar convergencia ni condiciones de parada).

Tanto en la figura 5.7 como en la 5.8 se aprecia que para tamaños muy pequeños de matrices, la potencia de cálculo de la GPU no es suficiente para ocultar el tiempo de transferencia CPU-GPU, aunque esta situación se invierte conforme aumentan las dimensiones. Sin embargo, a partir de 2048×2048 se vuelve a producir este efecto ya que se sobrepasa la capacidad de memoria de la GPU y aumenta, por tanto, el número de transferencias.

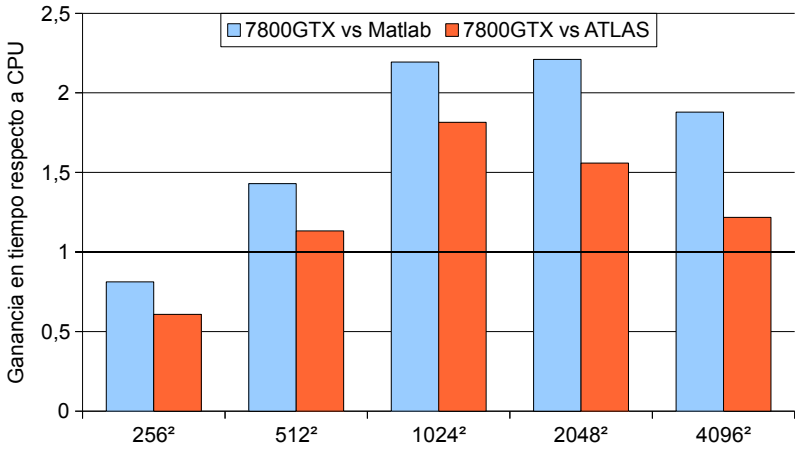

Figura 5.8: NMF-OpenGL

Ganancia en tiempo de ejecución de la GPU 7800GTX, respecto de `Matlab` y ATLAS.

Aunque con esta implementación se obtuvieron ganancias de tiempo más significativas respecto a la versión anterior, existen importantes limitaciones que merman su rendimiento. Por ejemplo, la memoria de texturas posee una geometría 2D, lo que obliga a que todas las matrices no solo deban caber en cuanto a su tamaño, sino también en el *número de filas y de columnas*. En ausencia de un gestor de memoria, es necesario reservar el espacio *en un orden determinado* para reducir al máximo la fragmentación. Además, cualquier transformación en la geometría de una matriz —por ejemplo, transponerla— no puede simularse cambiando el patrón de acceso a memoria, sino que dicha operación debe realizarse de manera explícita transfiriendo los datos a CPU y volviéndolos a copiar en la GPU en una nueva región de memoria.

5.1.2.4. Implementación en CUDA

Esta versión se basa en el modelo de programación *CUDA* [98], de NVIDIA, que representa una GPU como un coprocesador de propósito general capaz de ejecutar pequeñas porciones de código secuencial, denominadas *kernels*, en cientos o miles de hilos de manera simultánea. Para ello, CUDA proporciona una extensión del lenguaje C y una biblioteca de funciones que permite 1) reservar memoria en la GPU, 2) transferir datos entre la GPU y la CPU, y 3) ejecutar los mencionados *kernels*. Una descripción más detallada de este paradigma se realiza en el apéndice A.3.

Este modelo permite explotar un paralelismo de grano mucho más fino que otras tecnologías de procesamiento paralelo. Además, simplifica enormemente la implementación en GPU de algoritmos no relacionados con el dibujado de gráficos en pantalla [102]. No obstante, como se mencionó en anteriormente, muchos de estos dispositivos dis-

ponen de una memoria de capacidad limitada, lo que impone un procesamiento por bloques.

Por ello, se aplica un esquema similar al de la figura 4.4. Nótese que a diferencia de la versión en `OpenGL`, no se considera que las matrices **W** y **H** adquieran dimensiones significativas, por lo que pueden permanecer residentes en la memoria de la GPU. Por tanto, únicamente **V** es transferida por bloques, lo que simplifica la implementación del algoritmo y reduce drásticamente el uso del bus de datos.

Los productos de matrices se realizan utilizando la biblioteca de funciones *CUBLAS* [98], mientras que el resto de operaciones son llevadas a cabo por los *kernels* implementados específicamente. Al respecto, esta implementación parece obtener mejores resultados al procesar las matrices como memoria lineal y disponer los hilos CUDA de manera contigua en memoria. La idea es garantizar que todos los hilos de cada *warp* (unidad de planificación de hilos CUDA) accedan a direcciones de memoria consecutivas. En el caso de operaciones que dependen del número de columnas de las matrices, se utilizan posiciones de relleno para imponer dimensiones múltiplo de 16 (valor equivalente a medio *warp*) o 32 según la arquitectura. Mantener, de esta manera, los datos alineados compensa con creces el gasto adicional de memoria y los cómputos irrelevantes, ya que permite unificar diversas peticiones de acceso a memoria en una única operación.

También es importante resaltar que la matriz **H** se guarda en memoria *por columnas* (transpuesta). De esta manera, **W** y **H** tienen el mismo *"ancho"* (k) permitiendo utilizar los mismos *kernels* en ambas reglas de actualización. Por último, las transferencias se realizan de forma asíncrona, lo que permite solaparlas con la ejecución de los *kernels*. La concurrencia, en estos casos, se gestiona mediante *Streams* y *Eventos* de CUDA.

5.1.2.5. Pruebas y resultados en CUDA

A continuación se presentan los resultados de las pruebas de rendimiento con esta versión. Para mantener la coherencia con las medidas anteriores, sobre todo en lo referente al uso de arquitecturas *relativamente* contemporáneas, se ha empleado el procesador de gráficos *NVIDIA GeForce 8800GTX*, cuyas características aparecen en la tabla 5.2, y la versión de CUDA `0.8 beta`.

En la figura 5.9, se presentan los resultados de las pruebas realizadas comparándose con los tiempos de ejecución del resto de implementaciones (`Matlab`, ATLAS y GPU 7800GTX), tomados de la figura 5.7. Análogamente, la ganancia en tiempo se muestra en la figura 5.10.

Cuadro 5.2: Características de la GPU 8800GTX.

Marca y modelo	*NVIDIA GeForce 8800GTX*
Año	2007
Arquitectura	G80
Bus	*PCI Express*
Memoria de vídeo	768 MiB
Reloj interno	575 MHz
Reloj memoria	900 MHz
Interfaz memoria	384-bitshape
Ancho banda	86.4 GiB/s
Número de multiprocesadores	16
Cantidad de procesadores por multiprocesador	8
Relleno texturas	36800 MTexels/s

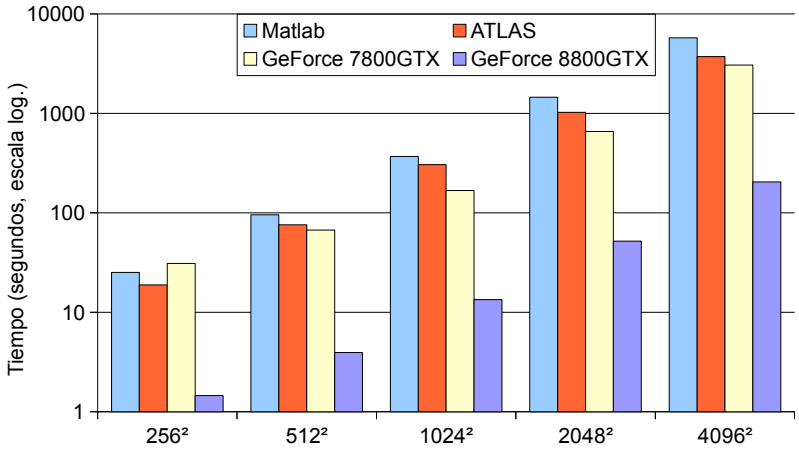

Figura 5.9: NMF-CUDA

Tiempos de ejecución del algoritmo de NMF implementado con `Matlab`, ATLAS y las GPU 7800GTX y 8800GTX, para matrices de distintas dimensiones. Pruebas realizadas con $k = 64$ y 2000 iteraciones (sin evaluar convergencia ni condiciones de parada).

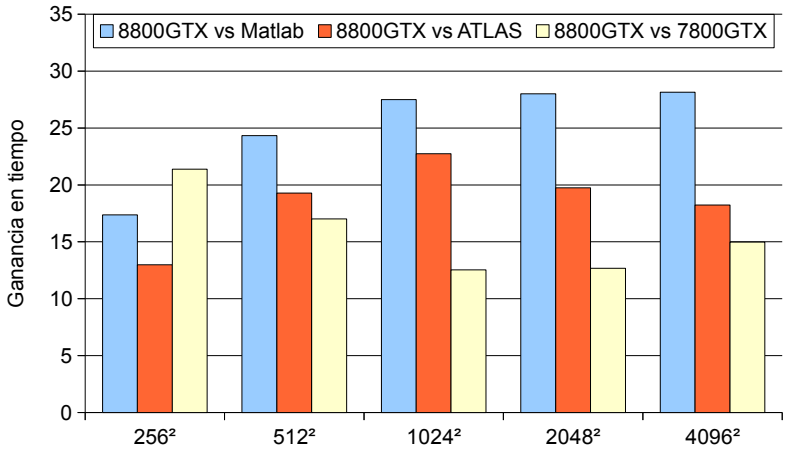

Figura 5.10: NMF-CUDA

Ganancia en tiempo de ejecución de la GPU 8800GTX, respecto `Matlab`, ATLAS y la GPU 7800GTX.

Los resultados que se aprecian son bastante importantes. Si bien ambas tarjetas son distintas, la mejora en las especificaciones técnicas no es suficiente para explicar una ganancia de tiempo tan importante. Parece, entonces, que CUDA aprovecha de una manera muy eficiente los recursos de la GPU. En especial, en lo concerniente a la implementación del producto de matrices (esto es, las bibliotecas CUBLAS) y en la ausencia de las limitaciones existentes en el otro modelo (tamaño de los *kernels*, ausencia de instrucciones de salto, etc).
No obstante, se nota un descenso en el rendimiento para la mayor de las matrices. Esto se debe a que se sobrepasa el límite de memoria disponible y por tanto, aumenta el número de transferencias CPU-GPU. Aun así, la ganancia en tiempo se mantiene importante.

5.2. Paralelismo de grano medio

Este nivel intermedio, de grano más grueso, corresponde a la distribución de los datos entre diversos elementos de cómputo; por ejemplo, un sistema multiprocesador o con varios dispositivos GPU. De esta manera, todos los elementos trabajan simultáneamente, cada uno aplicando alguna de las tecnologías del nivel inferior al subconjunto de datos que le es asignado. Se trata, por tanto, de la paralelización del bucle en la figura 4.4.

En la implementación de este nivel se ha utilizado la técnica del ***Paso de mensajes*** (*Message Passing Interface* o ***MPI***) para la sincronización de los diferentes elementos de cómputo. Como se aprecia en la figura 5.11, cada proceso se encarga de actualizar un conjunto de b_m *columnas* de $\mathbf{H}$, es decir, un bloque de dimensiones $k \times b_m$. Para ello, se necesitan las correspondientes b_m *columnas* de $\mathbf{V}$ y *toda* la matriz $\mathbf{W}$. Un esquema muy similar se utiliza para procesar la matriz $\mathbf{W}$, aunque en este caso se trata de bloques de b_n *filas* ($b_n \times k$). Para actualizarlos, son necesarias las correspondientes b_n *filas* de $\mathbf{V}$ y *toda* la matriz $\mathbf{H}$.

Dado que cada proceso MPI requiere una copia completa de las matrices $\mathbf{W}$ y $\mathbf{H}$, es necesario sincronizarlas tras cada actualización con el fin de mantener la coherencia entre todas las réplicas. En el caso de dispositivos GPU discretos —conectados al sistema mediante un bus de datos— tales operaciones conllevan, además, transferencias entre la memoria de la GPU y la memoria principal. Afortunadamente, el tamaño de estas matrices es bastante más pequeño que el de $\mathbf{V}$, con lo que sigue habiendo una reducción global de las transferencias.

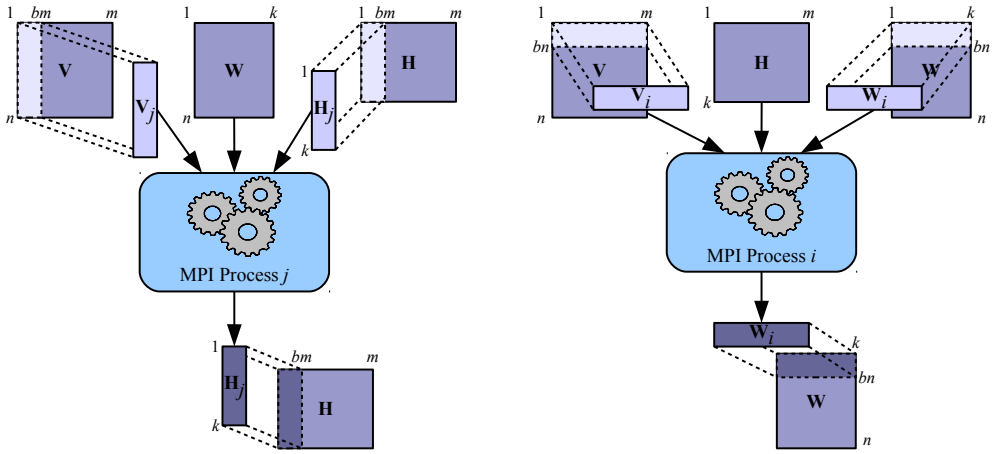

Figura 5.11: MPI: Esquema de partición de datos del algoritmo de NMF

Por último, se resalta que de manera similar a la implementación en CUDA, la matriz **H** se encuentra guardada en memoria *"por columnas"*, es decir, de manera transpuesta. Así, se evita un continuo proceso de empaquetado y desempaquetado de las columnas de **H** durante las comunicaciones.

5.2.1. Pruebas y resultados

A continuación se presentan los resultados de una implementación multi-GPU de NMF (MPI+CUDA). No obstante, para realizar una comparativa precisa del aporte de cada tecnología, y dado que tanto la plataforma como los datos de prueba son distintos a los empleados en el nivel inferior, a continuación se muestran los resultados de las cuatro combinaciones de estos dos niveles, a saber: CPU, CUDA, MPI (CPU) y MPI+CUDA.

Los experimentos se llevaron a cabo en las siguientes plataformas:

- Sistema Multiprocesador con 1036 nodos IBM BladeCenter JS20 (cada uno provisto de dos procesadores *IBM PowerPC 970FX* de 2,2 GHz y 4 GB de RAM DDR-PC2700), interconectados por una red *Myrinet* de alto rendimiento. La sincronización entre los procesos se realizó utilizando `mpich v1.2.6`, compilado con `xlc`. Por último, los productos de matrices se ejecutan a través de llamadas a la biblioteca de funciones `ATLAS v3.6` [94].
- Un servidor de dos procesadores *Xeon Dual-Core*, equipado con cuatro GPUs *Tesla C1060* de NVIDIA. Cada dispositivo tiene 240 núcleos distribuidos en 30 multiprocesadores, 4 GB de memoria y una *Capacidad de cómputo de NVIDIA* de 1,3. Las tres implementaciones fueron compiladas con `gcc 4.4.5` y `CU-`

`DA/CUBLAS 3.2` en un Linux de 64 bits. Para la sincronización se utilizó `mpich v1.2.7p1` con la configuración más parecida a la del sistema IBM.

Dado que el rendimiento de cada implementación depende en gran medida de las dimensiones de la matriz de entrada, hemos utilizado tres conjuntos de datos, procedentes de diversos estudios biológicos, que representan distintas clases de tamaño: pequeña, mediana y grande.

- *"ALL_AML"* (5000×38): Conjunto de 5000 genes analizados en 38 muestras de médula ósea correspondientes a dos tipos de tejido tumoral: *"Acute Lymphoblastic Leukemia"* (*ALL*) y *"Acute Myelogenous Leukemia"* (*AML*) [34].

- *"Cáncer"* (54675×1973): Nivel de expresión génica de 54 mil genes en 1973 muestras tumorales. Conjunto de datos procedente de la base de datos *GEO* [126] bajo el identificador '`GSE2109`'.

- *"HG"* (22283×5896): Análisis de más de 22 mil genes en 5800 muestras de tejido humano sano y afectado. Disponible en la base de datos *ArrayExpress* [127] bajo el identificador '`E-TABM-185`'.

A fin de homogeneizar todas las pruebas en la medida de lo posible, las matrices **W** y **H** se inicializan utilizando valores aleatorios procedentes de la misma *"semilla"*. Además, la ejecución del algoritmo queda fijada a 440 iteraciones, independientemente de la *"distancia"* entre **WH** y **V**.

En la tabla 5.3 se muestran los tiempos de cómputo de referencia, obtenidos con uno de los procesadores del *cluster* IBM. En las tres pruebas, se observa que los tiempos de ejecución crecen de forma lineal en función del *rango de factorización* (k).

Cuadro 5.3: Tiempos de ejecución, en segundos, de 440 iteraciones en la versión secuencial de base para diferentes rangos de factorización (k).

k	*ALL_AML*	*Cáncer*	*HG*
2	3.42	1842.29	2788.65
3	3.86	2069.99	2291.20
4	4.14	2404.44	2577.53
5	4.63	2397.94	2580.13
6	5.07	2620.29	2881.39
7	5.59	3131.00	3182.16
8	5.89	3147.86	3442.17
9	6.35	3124.68	3410.78
10	6.66	3348.15	3726.54

5.2.1.1. Rendimiento de la versión en MPI

Las figures 5.12 y 5.13 ilustran, respectivamente, los tiempos de ejecución de la mayor de las matrices (*HG*) y la Eficiencia media obtenida para todos los rangos de factorización (k).

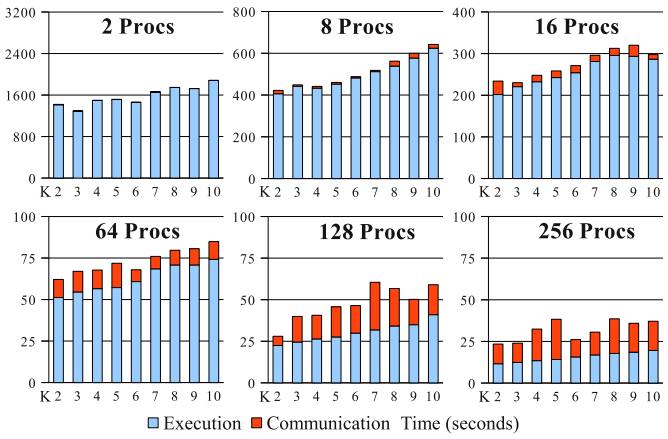

Figura 5.12: Versión
MPI: tiempos de cómputo y de comunicación entre procesos de la matriz *HG*, en segundos.

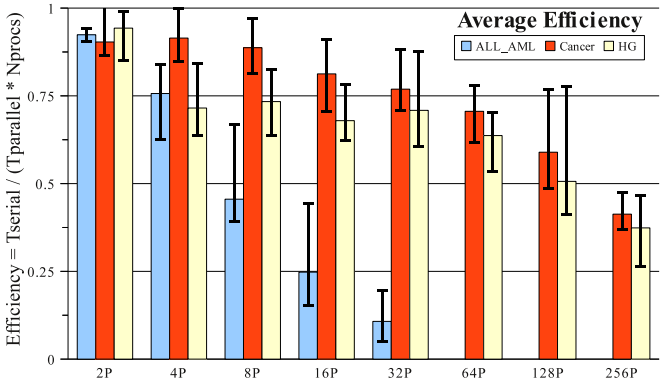

Figura 5.13: Versión MPI: Eficiencia
media obtenida para todos los rangos de factorización, en función del número de procesadores.

En matrices de medio y gran tamaño, las latencias producto de las comunicaciones entre procesos representan entre el 8 % y el 16 % del tiempo de ejecución con 32 CPUs, y aumentan gradualmente hasta alcanzar el 50 % con 256 procesadores. Esto representa una Eficiencia que decrece de 1,0 en el caso de dos procesadores, hasta 0,47 con 256 CPUs. Por su parte, en *ALL_AML*, la matriz de menor tamaño, estas latencias ocupan el 5 % del tiempo con dos procesadores, pero se incrementan rápidamente hasta el 70 % con sólo 16 CPUs.

A pesar de la reducción en el tiempo de cómputo, la escalabilidad está supeditada a

que haya suficiente carga computacional y un bajo coste en las comunicaciones entre procesos.

5.2.1.2. Rendimiento de la versión en GPU

La capacidad de memoria de la GPU utilizada es más que suficiente para albergar cualquiera de las matrices de prueba. No obstante, este podría no ser el caso en muchas arquitecturas. Para estudiar el impacto en el rendimiento que tendría tal situación, se ha simulado que la GPU tenga menos de 800 MB de memoria, lo que impone un procesamiento por bloques de la matriz *HG*.

La figura 5.14 muestra los tiempos de ejecución y de transferencia de datos de la versión GPU. Por su parte, en la figura 5.15 se indica el *speedup* respecto a la versión MPI. Lo primero que se observa es la regularidad en los tiempos de cómputo respecto del parámetro k. Esto se debe a las posiciones de *relleno* para garantizar el alineamiento de los datos en memoria. En cuanto al rendimiento, en el caso de *ALL_AML* se alcanza un valor bastante modesto (menos de 6x respecto de la versión base), ya que no existe suficiente carga computacional para mantener a miles de hilos CUDA ejecutándose concurrentemente. Caso contrario al de la matriz *Cáncer*, donde se obtiene un pico de 38x. Sin embargo, con *HG*, el rendimiento cae a 29x debido al procesamiento por bloques, cuyas transferencias ocupan el 30 % del tiempo de ejecución. Por último, se aprecia que con matrices medianas o de grandes dimensiones, la versión MPI necesita al menos 32 procesadores para superar el rendimiento de un único dispositivo GPU.

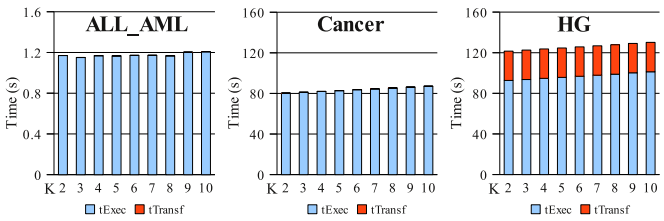

Figura 5.14: Versión GPU: tiempos de cómputo y de transferencia de datos, en segundos.

5.2.1.3. Rendimiento de la versión MPI-GPU

Al igual que en la versión anterior, hemos limitado la capacidad de memoria de los dispositivos. Las figuras 5.16a y 5.16b muestran, respectivamente, los tiempos de ejecución en *HG* y la ganancia de rendimiento respecto de la versión *mono-GPU*. Por

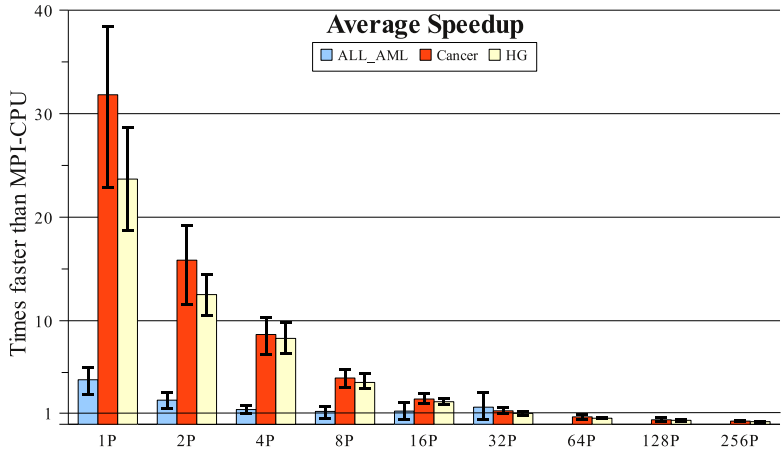

Figura 5.15: Versión GPU: *Speedup* medio respecto de la versión base ('1P') y MPI.

su parte, en las figuras 5.17a y 5.17b se aprecia la Eficiencia obtenida y el *Speedup* medio respecto de la versión base.

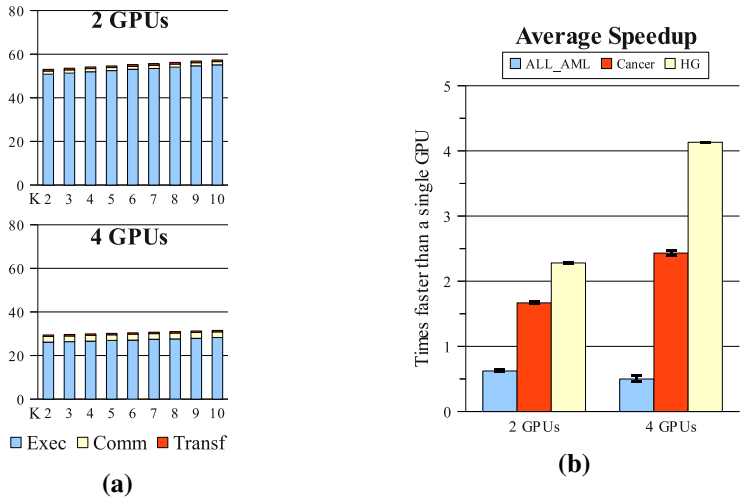

Figura 5.16: Versión MPI-GPU: (a) tiempos de cómputo, transferencias de datos y de comunicación entre procesos, de *HG*. (b) *Speedup* medio respecto de la versión *mono-GPU*.

Como era de esperar, el rendimiento de *ALL_AML* es bastante bajo. Las latencias por transferencias de datos y comunicaciones entre procesos suponen el 50 % del tiempo de ejecución. En cuanto a la matriz *Cáncer*, las transferencias son despreciables (3 % del tiempo), pero las comunicaciones representan el 7 % y el 17 % en dos y cuatro GPUs, respectivamente, lo que provoca una caída en la Eficiencia de 0,83 a 0,61. Finalmente, con *HG* se obtiene un ***speedup superlineal*** de 2,3x en dos GPUs, ya que las porciones asignadas a cada dispositivo no superan sus respectivas capacidades de memoria y pueden, por tanto, transferirse en una única operación al inicio del algoritmo. En el caso de cuatro GPUs sigue obteniéndose una ganancia *superlineal* (4,13x), pero la Eficiencia decrece ligeramente debido a las comunicaciones, que suponen el 9 % del tiempo.

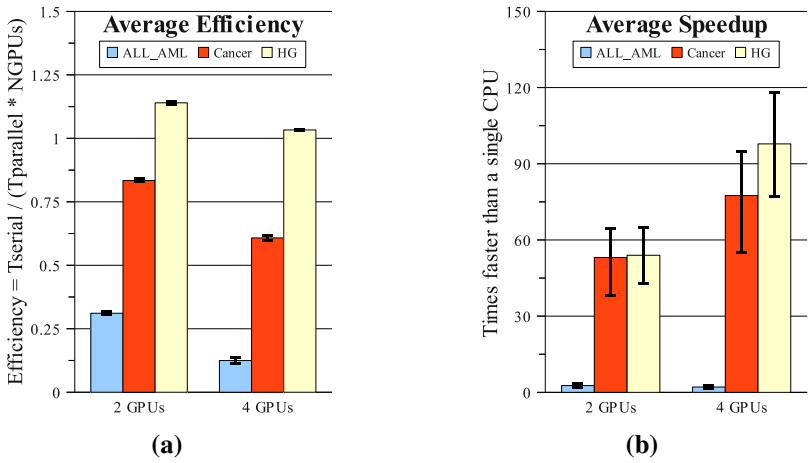

Figura 5.17: Versión MPI-GPU: (a) Eficiencia media respecto de la versión *mono-GPU*. (b) *Speedup* medio respecto de la versión base.

Si bien el *speedup* máximo respecto de la versión de base es de 120x, en la práctica, el rendimiento de esta implementación es comparable al de la versión MPI con 256 procesadores. En cuanto a la Eficiencia respecto de la versión *mono-GPU*, esta alcanza su valor más alto cuando se utiliza el *mínimo* número de GPUs que evita el procesamiento por bloques de las porciones asignadas a cada dispositivo. A partir de allí, las latencias por comunicaciones y transferencias de **W** y **H** comienzan a cobrar importancia.

5.3. Paralelismo de grano grueso

El nivel superior, de mayor granularidad, representa a todas las instancias de NMF que son ejecutadas para determinar el mejor rango de factorización. Dado que no existen dependencias de datos entre ellas, es posible distribuirlas entre varios sistemas de cómputo (por ejemplo, entre diversos *clusters* de GPU), donde cada uno aplique las técnicas de los niveles inferiores.

Para implementar este nivel, se desarrolló una nueva versión de NMF capaz de aprovechar las ventajas de otra tecnología reciente: las redes ***Grid***. Este tipo de entornos permiten coordinar recursos geográficamente distribuidos y ofrecer un acceso consistente y económico independientemente de su localización física o punto de acceso. De este modo se logra su uso como un único recurso para resolver aplicaciones intensivas en computación y/o datos a gran escala.

A continuación se expone una breve introducción a esta tecnología.

5.3.1. Cómputo en redes *Grid*

Una red ***Grid*** es un sistema distribuido de compartición de recursos informáticos de diversos tipos (unidades de cómputo, espacio de almacenamiento, programas, datos, etc.) que pueden encontrarse en puntos geográficos distantes. El término proviene de la analogía establecida por Ian Foster y Carl Kesselman en 1998 [128] con una *red de energía eléctrica*, en la que se proporciona un servicio —electricidad— de manera transparente y sin que el receptor tenga conocimiento de ciertos detalles como el lugar de origen (la central eléctrica que le surte) o el modo en que se genera.

Si bien la tecnología *Grid* se encuentra lejos de ese objetivo, actualmente permite la utilización de recursos compartidos entre diversas instituciones, en un entorno seguro y sin necesidad de que los miembros que aportan tales recursos deban modificar sus políticas internas de seguridad y de administración de sistemas.

Para ello, se emplea un conjunto de herramientas, denominado ***middleware***, que proporciona todos los servicios relativos al manejo de estos recursos: lanzamiento y ejecución de operaciones de cómputo, transferencia de datos, obtención de información sobre los recursos disponibles, etc. Este nombre se debe a que tales herramientas actúan como una *"capa intermedia"* entre el Sistema Operativo (o los servicios locales de administración de sistemas) y las aplicaciones de usuario, abstrayendo a estas últimas de todos los detalles referentes a las mencionadas operaciones. En la figura 5.18 aparece un esquema muy básico de este modelo de arquitectura.

Además del uso de *middlewares*, las instituciones y comunidades de usuarios participantes conforman las denominadas ***Organizaciones virtuales*** (*Virtual Organizations* o ***VO***), mediante las que establecen las políticas que rigen la utilización de sus recursos [129]. Por su parte, los usuarios acceden a la infraestructura *grid* identificados con un *certificado digital* emitido por alguna Entidad Certificadora que sea reconocida por todas las VO.

Es importante resaltar que una infraestructura *grid* debe poseer las siguientes características [130]:

- Gestión *distribuida* de los recursos, pues un manejo centralizado impediría la independencia de las políticas de administración de cada VO.
- Empleo de *estándares abiertos*, garantizando así la interoperabilidad entre los diferentes sistemas de cada organización, tanto los actuales como los futuros. Igualmente, facilita la evolución de esta tecnología y su adopción por parte de nuevos miembros.

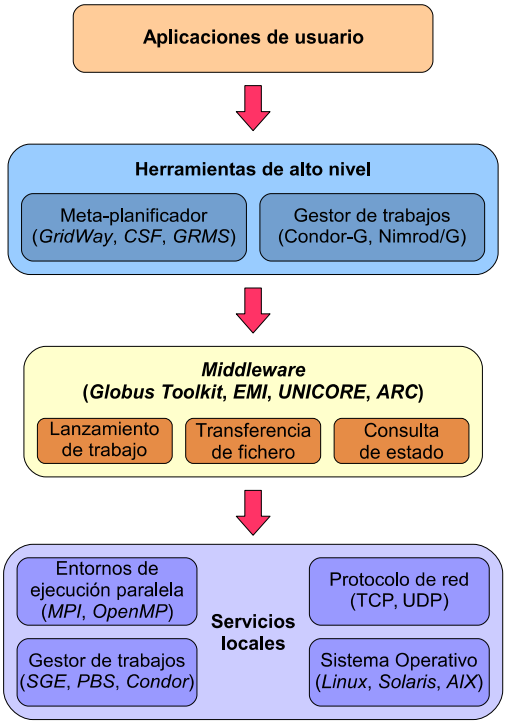

Figura 5.18: Uso de *middlewares* en infraestructuras *grid*

El *middleware* proporciona las herramientas para realizar operaciones básicas en una infraestructura *grid*, en la que puede existir una gran variedad de gestores locales de recursos. Por encima del *middleware* suelen utilizarse otras herramientas de alto nivel para facilitar, automatizar y proporcionar robustez a los servicios del *grid*.

- Distintas *calidades de servicio*. Es decir, que existan perfiles y roles de usuarios con diferentes niveles de acceso, tiempos de respuesta, etc., de manera que la utilización de los recursos sea *ordenada*.

Así, existen diversas implementaciones de *middlewares*, tales como *Globus Toolkit* [131] (el estándar *de facto* [130]), UNICORE [132], ARC [133] y EMI [134] (sucesor de *gLite* [135]). Por el contrario, ciertos gestores de colas, como OpenPBS [136] o SGE [137], *no* constituyen una infraestructura *grid*, puesto que tienen una gestión totalmente centralizada de los recursos. Igualmente, otros gestores, como Condor [138], no utilizan estándares abiertos ni permiten la interoperabilidad entre organizaciones que tengan distintas políticas de seguridad y/o de administración de sistemas.

Los servicios que proporciona un *middleware* son —no obstante— bastante básicos, de manera que el usuario debe gestionar manualmente el lanzamiento y monitorización de los trabajos, así como la descarga de los resultados. Esta tarea puede ser algo tediosa y propensa a fallos en un entorno donde la disponibilidad de los recursos es bastante dinámica. Por ello, tal y como se aprecia en la figura 5.18, por encima del *middleware* se suelen emplear herramientas de más alto nivel que faciliten el uso de la infraestructura y

añadan robustez. Un ejemplo de ello es un ***meta-planificador***, que permite una gestión automática de los recursos (desde el punto de vista del *usuario*) y de los trabajos lanzados. Así, si un trabajo falla (por ejemplo, debido a la falta súbita de un recurso), el meta-planificador es capaz de migrar o relanzar dicho trabajo automáticamente. Algunos ejemplos de meta-planificadores son *GridWay* [139], CSF [140] o GRMS [141], entre otros.

A continuación se describe un modelo genérico de la arquitectura de una infraestructura *grid*.

5.3.1.1. Arquitectura *grid*

En términos generales, la arquitectura *grid* está conformada por un modelo de varias capas de abstracción. De esta manera se asegura la interoperabilidad entre sistemas diferentes, así como el resto de características descritas previamente. La figura 5.19 muestra un esquema de este modelo.

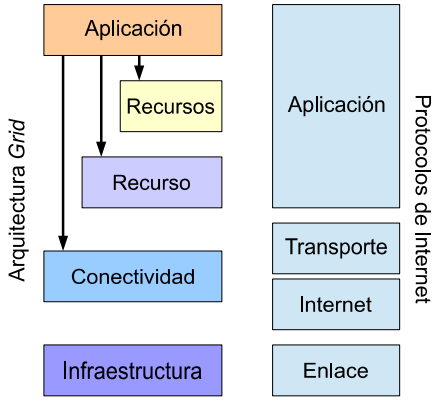

Figura 5.19: Esquema de la arquitectura *grid* (fuente: [129]).

Modelo por capas de la arquitectura *grid* y su correspondencia, aproximada, al conjunto de protocolos de Internet.

Las capas que lo conforman son las siguientes [129]:

Infraestructura (*Fabric*): interfaz a los recursos locales

Esta capa la componen todos los recursos, *físicos* y *lógicos*, que se van a compartir: unidades de cómputo, almacenamiento, transmisión de datos (infraestructura de red), programas, bases de datos, etc. En ella se implementan las operaciones específicas para dar servicios de acceso y utilización a cada uno de estos recursos: lanzamiento y monitorización de trabajos, lectura y escritura de ficheros, gestión y control de la red local, consulta a las bases de datos, etc.

Conectividad: comunicaciones en un entorno seguro

El objetivo de esta capa es proporcionar seguridad a todas las operaciones llevadas a cabo en el *grid*, tanto en lo referente a transacciones entre componentes remotos del nivel inferior, como a la autenticación —mediante certificados— de usuarios y organizaciones. Un aspecto importante a tener en cuenta en estos mecanismos es permitir la *coexistencia e independencia* de las políticas internas de seguridad y administración, de las VO integrantes.

Recurso: gestión de recursos individuales

Esta capa incluye todos los protocolos para el acceso remoto a *recursos individuales* a través de los mecanismos de seguridad que ofrece la capa inferior.
Se dividen en dos grandes categorías:

- *Protocolos de información*. Permiten obtener características del recurso, tales como su configuración, estado y carga actual.
- *protocolos de gestión*. Se encargan de iniciar las operaciones de acceso al recurso en función de las políticas de seguridad que sean aplicables.

Recursos (*Collective*): coordinación de conjuntos de recursos

Esta capa se encarga de proporcionar los servicios necesarios para la gestión de *grupos de recursos*.
Entre ellos, es posible destacar:

- Búsqueda de directorio. Permite conocer las características de alguna VO y los recursos que comparte.
- Planificación, asignación, monitorización y diagnóstico de recursos.
- Gestión de datos.
- Herramientas de alto nivel basadas en tecnología *grid*.
- Repositorios de programas.
- Servicios de contabilidad.

Aplicaciones

Esta última capa corresponde a todas las aplicaciones de usuario que operan dentro de una VO. Según sus necesidades, pueden acceder a servicios de las capas intermedias.

5.3.2. NMF en redes Grid

Esta versión se implementó utilizando el *middleware gLite* [135]. Como se muestra en la figura 5.20, el algoritmo lanza tantos trabajos como instancias de NMF sean necesarias para que se ejecuten simultáneamente en distintos sistemas de cómputo.

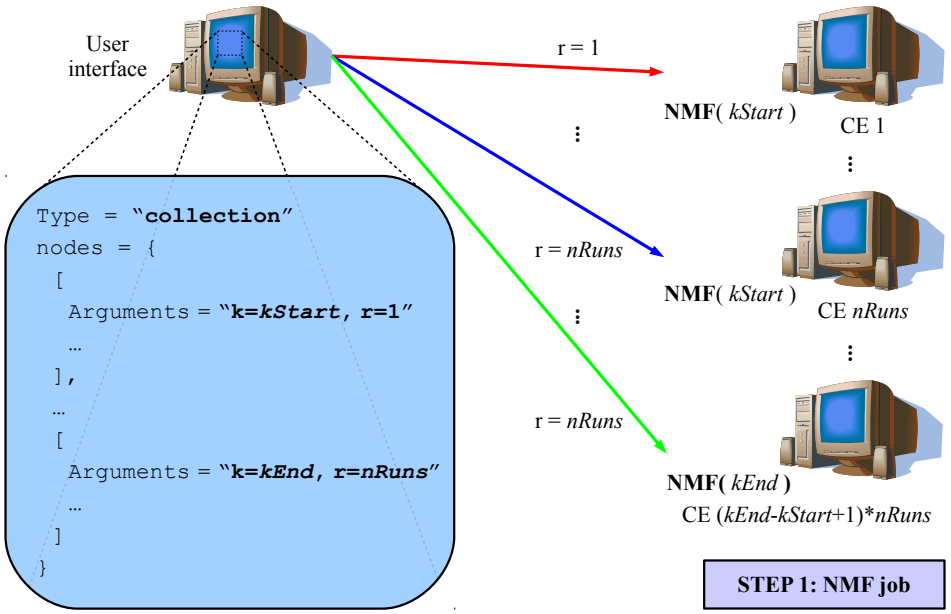

Figura 5.20: Esquema de NMF en redes de tipo *Grid*

El proceso es el siguiente:

1. Se transfiere la matriz de entrada a un nodo de almacenamiento (*Storage Element*, SE), pues esta podría ser demasiado grande como para enviarla adjunta en los trabajos.

2. Se lanzan todos los trabajos necesarios en función de los parámetros de entrada. En realidad, como se aprecia en la figura 5.20, se trata de un envío de tipo *"Collection"* que permite especificar en un único fichero JDL los parámetros de todas las tareas individuales.

3. En cada nodo de cómputo (*Computing element*, CE) se descarga el fichero de entrada y se ejecuta la factorización NMF con los parámetros correspondientes. Al acabar la ejecución, se transfieren los ficheros de salida al nodo de almacenamiento. Entre estos se encuentran las matrices **W** y **H**, así como otros conjuntos de datos temporales.

4. Al acabar todas las tareas, se descargan del nodo de almacenamiento todos los ficheros generados.

Capítulo 6

Arquitectura y organización del sistema

En este capítulo se detallan algunos aspectos globales de la aplicación, tales como la organización y la infraestructura sobre la que reposa el sistema, así como una descripción de su interfaz.

6.1. Entorno de ejecución

La nueva aplicación está orientada a su uso en las instalaciones informáticas presentes en los laboratorios de investigación. No obstante, en este tipos de equipos, normalmente multiprocesadores, se suele emplear un gestor de colas para la distribución de los trabajos que allí se ejecutan. Esto obliga a que los algoritmos no se ejecuten directamente (modo *interactivo*) sino a través de *scripts*, esto es, se debe incluir en un fichero de comandos la orden para ejecutar el algoritmo indicándole los datos a procesar. Este fichero sería tomado por el gestor de colas y lo ejecutaría en alguno de las unidades de cómputo disponibles.

Dado que este procedimiento depende completamente del gestor de colas empleado, se han implementado diversos *scripts* específicos para algunos de los gestores más extendidos, tales como *SGE* [137] y *OpenPBS* [136].
En este apartado también se incluyen ficheros de comando para enviar trabajos a una infraestructura *Grid*. En este caso, se emplea el *middleware gLite*.

Por ultimo, el sistema también está dotado de un pequeño meta-planificador que permite enviar trabajos a través del *Grid* cuando exista una cierta carga en el sistema (*cluster*)

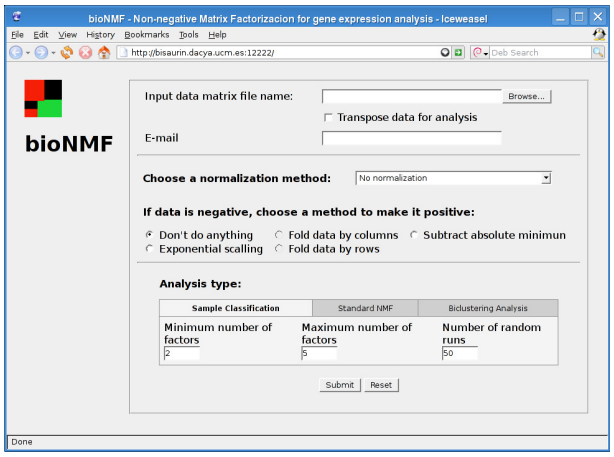

Figura 6.1: Interfaz web de la nueva aplicación.

local.

6.2. Interfaz de usuario

Otro de los objetivos de este proyecto, es proporcionar un interfaz que permita un fácil acceso a la aplicación, incluso a usuarios y equipos externos al laboratorio donde se encuentra instalado el sistema. Por ello, se ha dotado al mismo de dos tipos de interfaz que se describen a continuación.

6.2.1. Interfaz web

Se trata de una página web con un diseño visual parecido al de la aplicación bioNMF original (ver figura 6.1). En ella, el usuario puede enviar los datos que desea procesar sin necesidad de darse de alta en ningún tipo registro.

6.2.1.1. Utilización de la web

A continuación se describe brevemente el modo de empleo del interfaz web.

1. Especificar los datos a analizar (ver figura 6.1). El usuario debe indicar el fichero de datos, métodos de transformación y análisis a realizar, junto con sus respectivas opciones, de la forma indicada en la sección 3.1. Además, puede especificar una dirección de correo electrónico para que el sistema le notifique la finalización del análisis. Cuando el usuario pulsa el botón *submit*, se realiza una pequeña comprobación de los parámetros especificados y se cargan los datos en el servidor.

2. Una vez enviados los datos, el sistema muestra la página de la figura 6.2 a la espera de que el proceso finalice. En este punto, si el usuario indicó una dirección de correo, entonces puede cerrar la ventana. En cualquier caso, el sistema mantendrá la página hasta que el análisis haya finalizado.

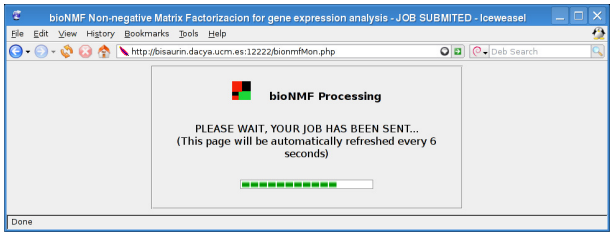

Figura 6.2: Interfaz web: página de espera.

3. Una vez finalizado el análisis, si el usuario especificó una dirección de correo electrónico, el sistema le notificará indicándole la dirección de la página de resultados. Esta página puede tener uno de los dos siguientes aspectos:

   - En caso de error, el sistema mostrará la página de la figura 6.3, indicando la causa del error producido.

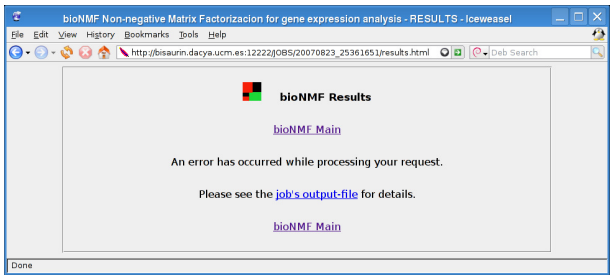

Figura 6.3: Interfaz web: página mostrada en caso de error.

   - En caso de éxito, el sistema muestra las gráficas resultantes y genera enlaces a las matrices obtenidas. A modo de ejemplo, se muestra la figura 6.4

6.2.1.2. Implementación

Este interfaz ha sido implementado utilizando *PHP* versión 5.2. Se trata de un lenguaje de programación para la generación dinámica de páginas web.

A continuación se describen brevemente los pasos llevados a cabo por el sistema durante el análisis de datos:

1. Crear un directorio único, de nombre aleatorio, para almacenar todos los ficheros que se generan.

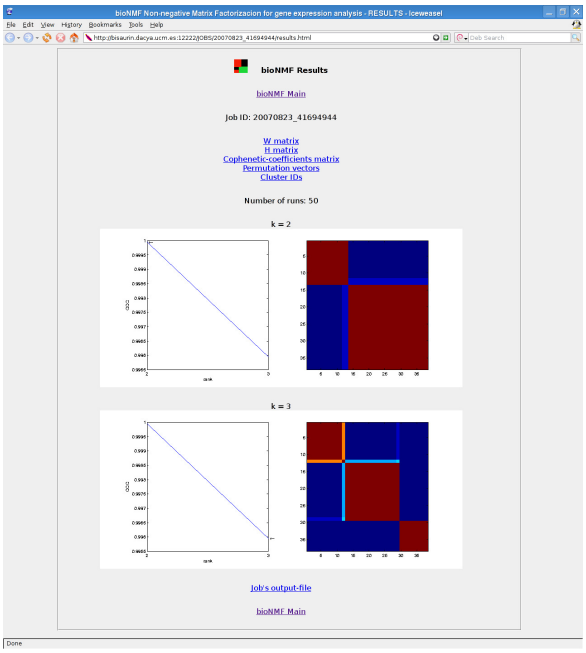

Figura 6.4:
Interfaz web: ejemplo de página de resultados para el algoritmo *Clasificación de ejemplares*.

2. Lanzar a ejecución, de forma independiente y en función del entorno de ejecución configurado en el sistema, el programa de análisis de datos.

3. Lanzar otro proceso, también independiente, encargado de generar la página de resultados (o de error) cuando el análisis haya acabado y de enviar un correo electrónico al usuario.

4. Mostrar la página de espera hasta que finalice el análisis.

5. Mostrar la página de resultados, o de error, generada por el segundo de los procesos.

Es importante resaltar que esta organización, en varios procesos independientes, permite que el usuario pueda cerrar la ventana de su navegador durante la página de espera sin que por ello el sistema se detenga, en especial en lo referente a la generación de la página de resultados/error y el envío del correo electrónico.

6.2.2. Interfaz de *Servicios web*

Frecuentemente los investigadores deben realizar un gran número de análisis sobre diversos tipos de datos, bien sea de forma secuencial, porque existen dependencias, o simultáneamente. Si bien el interfaz web descrito en el apartado anterior ofrece un fácil acceso a la aplicación, los pasos a seguir para lanzar cada análisis pueden convertir

este proceso en una labor tediosa, e incluso mermar la productividad del equipo de investigación.

Sería deseable, por tanto, disponer de un interfaz para aplicaciones (interfaz máquina-máquina) que permita automatizar este proceso.

De esta necesidad surgen los *Servicios web* (*Web services*). Se trata de un conjunto de estándares y protocolos para el intercambio de datos a través de la red entre aplicaciones que pueden estar implementadas en distintos lenguajes y/o ser ejecutadas sobre diversas plataformas.

El gran éxito de esta tecnología se debe en gran medida a la adopción de estándares abiertos para lograr la interoperabilidad, por ejemplo *XML* y *HTTP*. Prueba de ello, es que a menudo se trata simplemente de interfaces de aplicaciones web que *publican* funcionalidades en la red para ser utilizadas por otros programas o empresas.

En este sentido, la nueva aplicación *bioNMF* ofrece un interfaz que emplea esta tecnología. Este permite cargar la matriz de entrada en el servidor, ejecutar alguno de los métodos de análisis que se ofrecen y descargar los resultados.

Este sistema trabaja de forma asíncrona. Cuando el usuario transfiere la matriz de entrada, el servidor le devuelve un identificador que deberá utilizar como parámetro de entrada al momento de lanzar el análisis. Esta característica permite reutilizar la misma matriz de entrada en posteriores análisis que se quiera lanzar. Por su parte, cuando el usuario lanza un análisis, el servidor le devuelve un identificador del trabajo, con el que puede monitorizar el mismo y descargar los resultados cuando acabe.

Capítulo 7

Discusión y conclusiones

El desarrollo de técnicas de extracción de datos experimentales de alto rendimiento, como es el caso de los Chips de ADN (*DNA Microarrays*), ha supuesto la acumulación de enormes cantidades de información que la comunidad científica busca analizar. De allí que hayan surgido diversos campos de estudio, tales como la Bioinformática y la Biología Computacional, cuyo objetivo es el de proporcionar métodos y herramientas que permitan a los investigadores llevar a cabo esta tarea.

No obstante, ante el continuo crecimiento de la información experimental generada no es suficiente con disponer de la funcionalidad de tales técnicas de análisis, también es necesario que estas sean eficientes para no generar *cuellos de botella* en las labores de investigación. Un claro ejemplo de esta situación se presenta con la factorización *NMF* y los métodos basados en esta, *Clasificación de muestras* y *Agrupamiento doble*. A pesar de las claras ventajas que estos presentan para la extracción de información biológica a partir de grandes cantidades de datos experimentales, su utilidad o aplicación práctica ha ido mermando con el paso del tiempo debido a sus enormes necesidades en recursos computacionales.

Por ello, en este trabajo de tesis se ha abordado la optimización y paralelización de estos tres algoritmos. El primer paso ha sido analizar las operaciones que los componen e identificar los niveles o grados de paralelismo inherentes. Como resultado, se ha obtenido un modelo un modelo de paralelismo, por capas, que permite la aplicación selectiva o combinada de diferentes estrategias. Al respecto, el siguiente paso ha sido la búsqueda de tecnologías de alto rendimiento que aprovechen las estrategias asociadas a cada nivel.

Una de estas tecnologías ha sido la de las *Unidades de Procesamiento de Gráficos* (GPU). Si bien estos dispositivos fueron diseñados inicialmente para el dibujado de gráficos

en la pantalla, su enorme capacidad de cómputo (muy superior a la de procesadores convencionales) y en especial la posibilidad de ejecutar algoritmos de álgebra lineal, han despertado un gran interés en la comunidad científica para aplicaciones en gran variedad de campos de la Ciencia. Además, como ya están disponibles en multitud de dispositivos de uso general (PC de sobremesa, portátiles, sistemas empotrados como teléfonos móviles, tabletas, etc), representan una alternativa "de bajo coste" a los sistemas multiprocesadores clásicos (*clusters*).
No obstante, el modo de programación de estos dispositivos era en sus inicios bastante rígido y aún muy orientado a las aplicaciones gráficas (incluso era necesario utilizar lenguajes de síntesis de gráficos como `OpenGL`), lo que dificultaba la adaptación de algunos algoritmos de cómputo general a esta arquitectura. Es por ello que la primera versión de NMF en esta plataforma se obtuvo una ganancia de rendimiento, respecto a procesadores tradicionales, algo "modesta": $3X$ (tres veces más rápido que una CPU convencional).
Afortunadamente, tanto la arquitectura como el modelo de programación evolucionaron y la presentación de nuevos paradigmas, como CUDA, permitieron la utilización de las GPU como verdaderos coprocesadores de cómputo general. Al respecto, la siguiente implementación de NMF con este nuevo modelo alcanzó niveles de rendimiento bastante más acusados, con una ganancia de casi $40X$. Sin embargo, dado que este tipo de dispositivos suelen tener una memoria de uso exclusivo de muy poca capacidad, es necesario un procesamiento por bloques que incluye transferencias de datos, y que debe hacerse con cuidado para limitar en lo posible la perdida en el rendimiento.
Para resolver esta situación se desarrolló otra versión del algoritmo adaptada a sistemas multi-GPU. Esta configuración, aunque no reduce la cantidad total de transferencias, sí que permite realizarlas de manera simultánea, obteniendo así una *ganancia superlineal*; es decir, que dos dispositivos trabajando en conjunto tienen un rendimiento *superior al doble* que el obtenido con uno solo. Además, al combinar cuatro de estos dispositivos se obtiene un rendimiento *120 veces superior* al de un procesador tradicional.

Otra de las tecnologías analizadas en esta tesis ha sido la de redes tipo *Grid*. Este tipo de entorno permite coordinar recursos geográficamente distribuidos, ofreciendo un acceso consistente y económico, independientemente de su localización física o punto de acceso. Una de las características más importante de esta tecnología es que no es necesario que los miembros que aportan tales recursos deban modificar sus políticas internas de seguridad y de administración de sistemas.
Este tipo de infraestructura es ideal para cómputos de grano muy grueso y desacoplados. Es por ello que empleó para ejecutar la gran cantidad de instancias independientes de NMF, que conforman la técnica Análisis Exploratorio de Datos (EDA) utilizada en el

método Clasificación de Muestras.
Desafortunadamente no se contó con una infraestructura *grid* que tuviera disponible nodos de cómputo compuestos por las tecnologías empleadas en los niveles inferiores (por ejemplo, un sistema multi-GPU).

A pesar de la validez del modelo de optimización empleado y de las importantes ganancias en tiempo obtenidas en las pruebas de rendimiento, la utilidad de este trabajo se vería limitada si se queda en el plano teórico. En efecto, aunque existe un gran interés en la comunidad científica por el uso de tecnologías de alto rendimiento como las aquí mencionadas, su utilización suele requerir conocimientos avanzados en informática y administración de sistemas. El perfil medio de un investigador no suele incluir este tipo de formación. Además, que ello le restaría tiempo a las tareas de análisis que son las realmente importantes. Así, en esta tesis también se ha marcado como objetivo la construcción de una *herramienta* que ponga a disposición de la comunidad científica las técnicas de optimización empleadas en este trabajo.

La aplicación propuesta permite el procesamiento de conjuntos de datos *de gran tamaño* en equipos de *envergadura muy variada*: desde simples ordenadores portátiles y PC (supeditado a los recursos disponibles), hasta servidores y sistemas multiprocesadores de alto rendimiento. Además, incluye las dos *metodologías de análisis de datos de expresión génica* mencionadas anteriormente, así como algunos *métodos de preprocesamiento* que permiten normalizar y adaptar los datos a los requerimientos de la factorización NMF. También incluye dos interfaces que ocultan todos los detalles de la implementación: por un lado dispone de un interfaz web que facilita enormemente su utilización a cualquier investigador con conocimientos muy básicos de informática; por otro, un servidor de *servicios web* (*web services*) que permite un acceso automatizado a aquellos usuarios con más experiencia en programación. Es importante resaltar, que dichos interfaces posibilitan un acceso remoto, bien sea *público y anónimo*, o *restringido* a través de una red privada.

Para facilitar aún más el empleo de esta herramienta, además de la publicación del código fuente, se puso a disposición de la comunidad Bioinformática una versión de esta aplicación que se ejecuta en los servidores del Departamento. El acceso a la misma es público, gratuito y anónimo. Su puesta en marcha a través de la dirección http://bionmf.dacya.ucm.es, tuvo una buena acogida por parte de la comunidad científica, obteniendo una media de ***75 trabajos mensuales*** procedentes de distintos países. Además, algunas instituciones externas han solicitado el código fuente para adaptarlo a sus propias infraestructuras.

Si bien esta aplicación aún puede ser mejorada, como se verá a continuación, supone una excelente herramienta que ayuda al trabajo que a diario realizan los investigadores para extraer información biológica de enormes cantidades de datos experimentales.

Trabajo futuro

Existen diversas lineas de trabajo que permitirían ampliar la utilización y funcionalidad de esta herramienta. Por ejemplo, se pueden añadir más variantes y evoluciones del algoritmo NMF que han aparecido en los últimos años [49, 53, 54], algunas de ellas que mejoran en diversos aspectos al algoritmo NMF estándar.

Además, la implementación para dispositivos GPU presentada, en CUDA, restringe su uso a sistemas de NVIDIA. Al respecto, desde hace algunos años se ha propuesto otro lenguaje, `OpenCL` [101] que permite su utilización en plataformas multi-núcleo de otras empresas (ATI, Intel, etc).

Finalmente, también se puede considerar el uso de nuevas tecnologías y paradigmas como el *Cloud Computing*, para el que ya han ido apareciendo algunas implementaciones de NMF [142].

Apéndice A

Las Unidades de Procesamiento de Gráficos (GPU)

En este apéndice se describe el funcionamiento de las ***Unidades de Procesamiento de Gráficos*** (*Graphics Processing Units* o ***GPU***). Se trata de dispositivos diseñados específicamente para realizar todas las operaciones trigonométricas y de álgebra lineal que son necesarias para el dibujado de objetos 3D en una pantalla bidimensional.

A.1. Dibujado de gráficos en pantalla

La representación de un objeto tridimensional en una pantalla 2D requiere calcular su proyección sobre un plano. Para ello, es necesario conocer ciertos detalles, tales como:

- **Geometría del objeto** a mostrar por pantalla. Suele describirse mediante los ***vértices*** del conjunto de polígonos que conforman la malla con la que se le ha dado forma al objeto.
- **Iluminación ambiental** de la escena. Se emplea para determinar diferentes tonalidades de cada color, lo que creará la sensación de profundidad en la imagen resultante.
- **Punto de vista** del observador, con el que se calcula la perspectiva del objeto a representar.

Toda esta información es procesada por los dispositivos GPU a lo largo de varias etapas, en lo que se denomina ***tubería de gráficos*** (*graphics pipeline*), dando como resultado

la imagen 2D final. En la figura A.1 se muestra un esquema básico de todo el proceso.

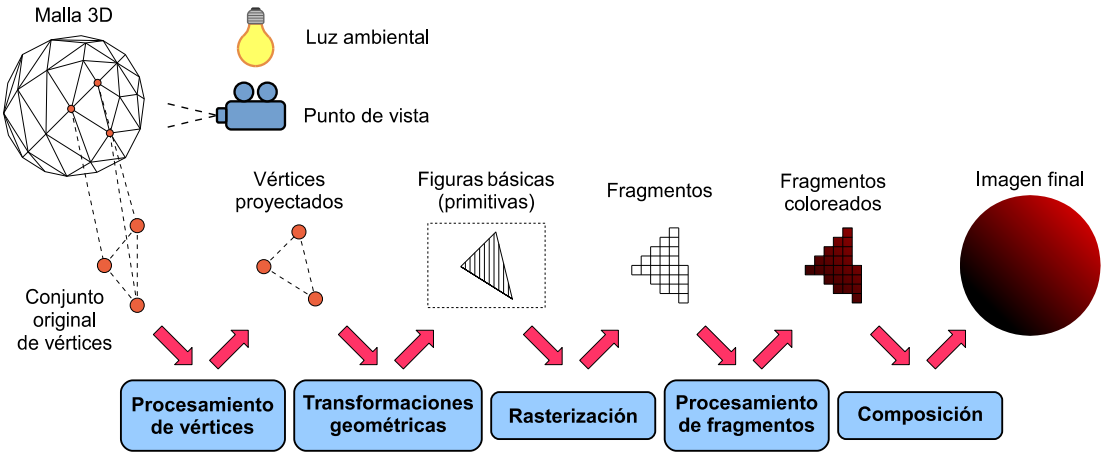

Figura A.1: Esquema básico de una *tubería de gráficos*

Conjunto de etapas en una GPU en las que se procesa un objeto 3D para representarlo en una pantalla bidimensional.

Estas etapas son las siguientes:

- **Procesamiento de Vértices:** en esta primera etapa se toma la información antes mencionada, además de otros aspectos como el color y el "material" (textura) del que estaría hecho el objeto a representar. Con todo ello se *proyectan las coordenadas de los vértices* según el punto de vista del observador y la iluminación ambiental.

- **Transformaciones geométricas:** consiste en agrupar los vértices en figuras geométricas básicas, denominadas ***primitivas***, tales como triángulos, cuadrados y otros polígonos.

- **Rasterización:** en esta fase se determinan los puntos de la pantalla, denominados ***fragmentos***, que quedarían contenidos dentro de cada *primitiva*. El resultado equivale a una *discretización* o muestreo de la figura primitiva.

- **Procesamiento de fragmentos:** consiste en colorear todos los fragmentos en función de la iluminación y la perspectiva calculada en las etapas anteriores.

- **composición**, todos los fragmentos son combinados para obtener la imagen final.

A.2. Evolución arquitectónica de los procesadores de gráficos

Inicialmente, todas las operaciones descritas en la sección anterior eran llevadas a cabo en el procesador principal del sistema (CPU), por lo que las ***tarjetas gráficas*** presentes en computadores personales (PC) y otros equipos informáticos, únicamente se ocupaban de representar la imagen 2D final. Sin embargo, ante el enorme e incesante crecimiento del mercado de videojuegos, en el que los objetos 3D son cada vez más complejos y detallados, todas estas operaciones se han ido trasladando fuera de la CPU, conllevando a la aparición y desarrollo de las *Unidades de Procesamiento de Gráficos* (*GPU*).

Una evolución importante en la arquitectura de estos dispositivos se obtuvo con el desarrollo de los ***procesadores de vértices*** y de ***fragmentos***, encargados de las respectivas fases de la "tubería de gráficos". Dado que en dichas etapas las operaciones se realizan de manera independiente para cada elemento, al añadir ***varias unidades de cómputo de cada tipo***, fue posible procesar diversos elementos de manera simultánea. Esto permitió una importante mejora del rendimiento que sobrepasó a la potencia de cálculo de las CPU convencionales de la época.

Otro hito importante fue la flexibilización de las acciones llevadas a cabo en todo el proceso. Esto es, inicialmente todas las operaciones eran totalmente *estáticas* (predeterminadas de fábrica), pero en generaciones posteriores de procesadores de gráficos se añadió la posibilidad de ejecutar ***funciones programadas por el usuario*** (*shaders*) en algunas de ellas, especialmente en los *procesadores de vértices y de fragmentos*. Esta característica es la que permitió a la comunidad científica comenzar a utilizar estos dispositivos para aplicaciones que nada tienen que ver con gráficos en la pantalla.

En la figura A.2, referente a la arquitectura *G70* de NVIDIA, quedan ilustrados estos dos últimos aspectos. En ella se observan las diversas unidades de cómputo programables para las etapas de procesamiento de vértices y de fragmentos.

A.2.1. GPU de arquitectura unificada

A partir de lo que se conoce como *quinta generación* de GPU, se produce un cambio radical en su arquitectura, ya que las diferentes unidades de cómputo quedan sustituidas por un mismo tipo de procesadores capaces de ejecutar todos los cómputos requeridos. Estos nuevos núcleos de cómputo (***cores***), cuyo número es considerablemente superior a los existentes en la generación anterior, aparecen distribuidos en varios conjuntos de

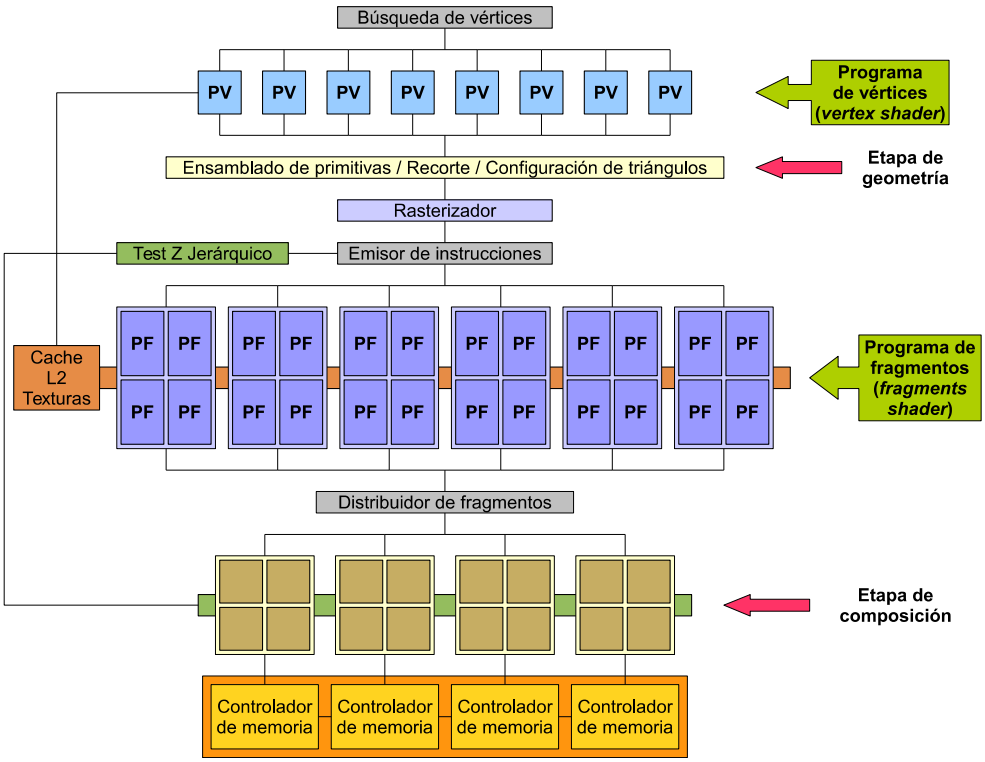

Figura A.2: Esquema simplificado de la arquitectura *G70* de NVIDIA (fuente: NVIDIA).

Posee 8 procesadores de vértices (PV) y 24 procesadores de fragmentos (PF). Además, estas unidades admiten la ejecución de funciones programadas por el usuario (*shaders*).

procesadores multi-núcleo. Esta organización permite la ejecución ***simultánea*** de una instrucción sobre datos distintos, lo que se conoce como ***SIMD*** (***Single-Instruction, Multiple-Data***), ocultando así las latencias de cada operación.

En la figura A.3 se muestra un esquema simplificado de la arquitectura *G80* de NVIDIA, en el que se aprecia la gran cantidad de núcleos de cómputo y su organización. Estos se han ido masificando en generaciones posteriores, además de incluir diversas mejoras en el acceso a memoria.

A.3. Cómputo en Dispositivos de Arquitectura Unificada (CUDA)

Con el desarrollo de la *Arquitectura Unificada* de NVIDIA, esta empresa también presentó un nuevo modelo de programación: ***CUDA*** (*Compute Unified Device Architecture*), que generaliza el concepto de GPU presentándolo como un *coprocesador de propósito general* que puede ser programado mediante una extensión del lenguaje C [98].

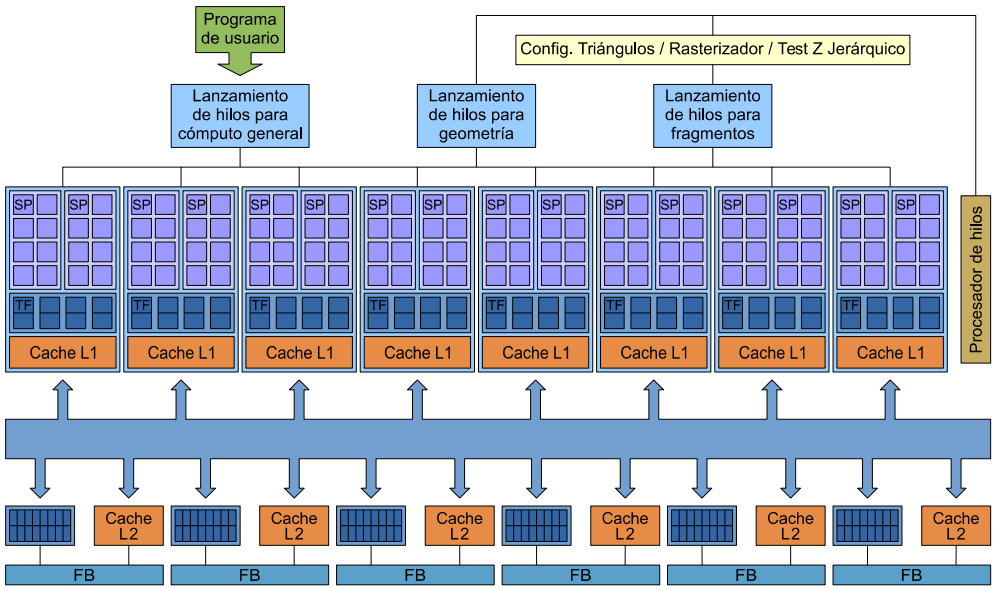

Figura A.3: Esquema simplificado de la arquitectura *G80* de NVIDIA (fuente: NVIDIA).

Posee 8 bloques con dos procesadores multi-núcleo, cada uno con 8 *cores* (marcados como *"SP"*), lo que representa un total de ***128 unidades de cómputo***.

Bajo este paradigma, la GPU es vista como un sistema compuesto por cientos de unidades de cómputo simples, denominadas ***cores***, que permiten la ejecución ***de manera simultánea*** de una instrucción sobre datos distintos (*single-instruction, multiple-data* o SIMD). Estas unidades funcionales se encuentran agrupadas en diversos multiprocesadores, cuyo número y distribución varía en función de la arquitectura.

Desde el punto de vista del programador, se contemplan dos ámbitos de ejecución: el del ***dispositivo*** (*device*), que abarca el código que se va a ejecutar en la GPU; y el del ***anfitrión*** (*host*), correspondiente a la CPU desde la que se ordena el lanzamiento de dicho código. Cada ámbito, además, está provisto de su propio espacio de memoria: la memoria principal para el anfitrión y la memoria de vídeo para el dispositivo. Por ello, la extensión del lenguaje C que proporciona CUDA contiene funciones para 1) reservar memoria en la GPU, 2) transferir datos entre la GPU y la CPU, y 3) ejecutar código en el dispositivo bajo la forma de pequeñas rutinas denominadas ***kernels***.

Cada *kernel* está compuesto por un reducido conjunto de *instrucciones secuenciales*, cuyo objetivo es únicamente procesar *unos pocos datos de entrada*. Así, al invocarlo desde el anfitrión, el programador debe especificar el número de veces que dicho *kernel* ha de ejecutarse para abarcar el conjunto total de datos. En ese momento, en la GPU se crean igual número de instancias del programa, denominadas ***hilos*** (*CUDA threads*), que intentan ejecutarse de manera simultánea en función de las unidades de cómputo disponibles.

A.3.1. Organización de los hilos de ejecución

En CUDA, el conjunto de hilos encargados de ejecutar un *kernel* se distribuyen, a partes iguales, en ***bloques de hilos***; y estos, a su vez, se organizan en una ***malla*** (*grid*) o cuadrícula de bloques. Por tanto, al momento de lanzar un *kernel* es necesario especificar las dimensiones de cada bloque de hilos y de la malla de bloques, tal como se aprecia en la figura A.4.

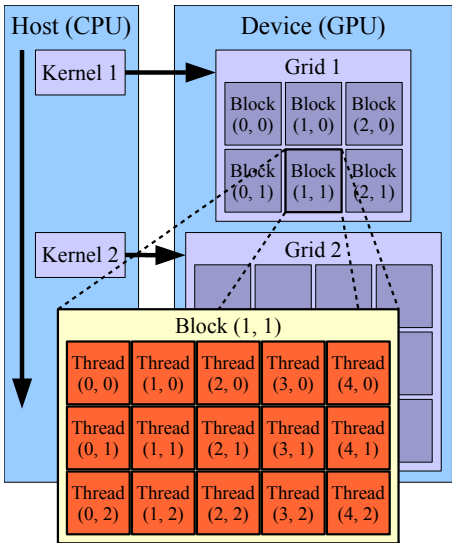

Figura A.4: CUDA: esquema de ejecución de *kernels* (fuente: [98]).

Cada hilo es representado por un identificador que coincide con su índice dentro del bloque. En caso de que el bloque sea de dos o tres dimensiones, se procede de la siguiente manera:

- Para un bloque de dimensiones (D_x, D_y), el identificador del hilo de índices (x, y) será $(x + yD_x)$.
- Para un bloque de dimensiones (D_x, D_y, D_z), el identificador del hilo de índices (x, y, z) será $(x + yD_x + zD_yD_x)$.

De forma análoga a los hilos, cada bloque posee un identificador que coincide con su índice dentro de la malla de bloques. Para mallas 2D o 3D, el identificador de cada bloque se calcula de la misma forma que para los hilos en bloques de dos o tres dimensiones.

Es importante resaltar que este modelo permite la suficiente flexibilidad para que un kernel pueda ejecutarse en dispositivos con distinta capacidad de cómputo en paralelo sin tener que recompilar su código, ya que un dispositivo con poca capacidad, ejecutará

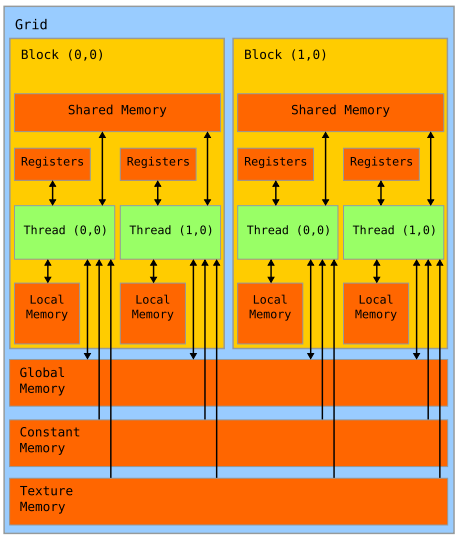

Figura A.5:
Modelo de memoria. Cada hilo de ejecución tiene acceso a la memoria del dispositivo a través de un conjunto de *espacios de memoria* de distinto ámbito. Fuente: *NVIDIA CUDA* [98].

los bloques secuencialmente, mientras que otro con mayor potencia lo hará de forma simultánea.

A.3.2. Modelo de memoria

Los hilos de ejecución sólo pueden acceder a la memoria DRAM del dispositivo y a la memoria interna del procesador a través de los siguientes espacios de memoria, ordenados de menor a mayor tiempo de acceso (ver, además, la figura A.5):

- *Registros* para cada hilo de ejecución.
- *Memoria Local*: accesible a cada hilo de ejecución.
- *Memoria Compartida*: accesible a todos los hilos de un bloque.
- *Memoria Global*: accesible a todos los hilos que pertenezcan a la misma malla de bloques.
- *Memoria de constantes*: memoria de sólo lectura accesible a toda la malla de bloques.
- *memoria de texturas*: memoria de sólo lectura accesible a toda la malla de bloques.

Es importante resaltar que la memoria global no posee memoria *cache* y que un patrón de acceso correcto (esto es, que evite conflictos de acceso en los bancos de memoria) es crucial para aprovechar todo el ancho de banda que ofrece. Por su parte, la memoria

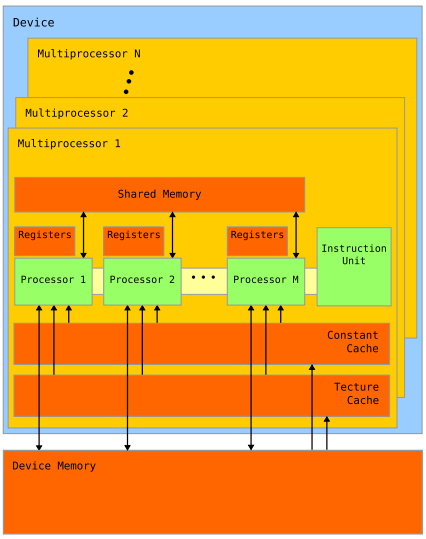

Figura A.6: Modelo arquitectónico. Conjunto de multiprocesadores con arquitectura SIMD. Fuente: *NVIDIA CUDA* [98].

de constantes sí dispone de memoria cache y en condiciones óptimas, puede ofrecer un tiempo de acceso tan rápido como el de un registro. En lo referente a la memoria de texturas, ésta también dispone de una cache y está optimizada para aprovechar la localidad 2D de los datos.

A.3.3. Modelo arquitectónico

El dispositivo está implementado como un conjunto de multiprocesadores (ver figura A.6), cada uno de los cuales está diseñado con una arquitectura SIMD (*Single Instruction, Multiple Data*), esto es, en cada ciclo de reloj, todos los procesadores de un multiprocesador ejecutan las mismas instrucciones pero sobre datos distintos.

Cada multiprocesador posee los siguientes tipos de memoria interna, integrada en el chip:

- Un conjunto de *registros* locales a cada procesador.
- Una cache de datos compartida por todos los procesadores. Implementa al espacio *Memoria compartida* mencionado en la descripción del modelo de memoria.
- *Cache de constantes*: una cache para datos de sólo lectura, también compartida por todos los procesadores, que reduce el tiempo de acceso al espacio de sólo lectura *Memoria de constantes*.
- *Cache de texturas*: cache de sólo lectura compartida por todos los procesadores y que reduce el tiempo de acceso al espacio *Memoria de texturas*.

Los espacios de memoria local y global están implementados por regiones de la memoria externa del dispositivo y no poseen una memoria cache. Por su parte, los espacios de constantes y de texturas antes mencionados, están también implementados como regiones de la memoria externa, aunque de sólo lectura.

A.3.4. Modelo de ejecución

El dispositivo procesa una malla de bloques ejecutando uno o más de estos bloques en cada multiprocesador mediante *time-slicing*. Cada bloque es entonces dividido en grupos de hilos denominados *warp*, teniendo cada uno el mismo número de hilos y siendo estos últimos ejecutados al estilo SIMD. Además, un planificador de hilos se encarga de cambiar periódicamente el *warp* que se encuentra en ejecución para maximizar el uso de los recursos del multiprocesador.

Un bloque es procesado por un único multiprocesador, de manera que el espacio *memoria compartida* reside en la memoria compartida integrada en el chip, lo que conlleva a accesos muy rápidos al mismo. Además, los registros del multiprocesador se distribuyen entre todos los hilos del bloque.

No existe un orden predefinido de ejecución de los *warps* pertenecientes a un bloque, aunque como se mencionó anteriormente, es posible sincronizar los hilos de un bloque para coordinar su acceso a las memorias global y compartida. Así mismo, tampoco existe un orden de ejecución de los bloques pertenecientes a una malla de bloques, pero en este caso no existe ningún mecanismo de sincronización entre los bloques y por tanto, no pueden “comunicarse” mediante la compartición de datos.

Bibliografía

[1] G. Stoesser, W. Baker, A. van den Broek *y cols.*: ***The EMBL Nucleotide Sequence Database***. *Nucleic Acids Res.*, 30(1):21–26, 1/2002.

[2] T. N. Bhat, P. Bourne, Z. Feng *y cols.*: ***The PDB data uniformity project.*** *Nucleic Acids Res.*, 29(1):214–218, 1/2001.

[3] J. Barker y J. Thornton: ***Software engineering challenges in bioinformatics***. En F. Titsworth (ed.): ***Proc. 26th Int. Conf. Softw. Eng.***, págs. 12–15, Los Alamitos, CA, 5/2004. *IEEE Computer Society*.

[4] P. O. Brown y D. Botstein: ***Exploring the new world of the genome with DNA microarrays***. *Nat. Genet.*, 21(Suppl 1):33–37, 1/1999.

[5] National Institutes of Health. National Human Genome Research Institute: ***Talking Glossary of Genetic Terms***. http://www.genome.gov/Glossary/ [03-08-2014].

[6] B. Alberts, A. Johnson, J. Lewis *y cols.*: ***Molecular biology of the Cell***. *Garland Science*, 5ª ed., 11/2007.

[7] A. Brazma y J. Vilo: ***Gene expression data analysis***. *FEBS Lett.*, 480(1):17–24, 8/2000.

[8] R. L. Stears, T. Martinsky y M. Schena: ***Trends in microarray analysis***. *Nat. Med.*, 9(1):140–145, 1/2003.

[9] Affymetrix Inc: **.** http://www.affymetrix.com [03-08-2014].

[10] S. Drăghici: ***Data Analysis Tools for DNA Microarrays***. Chapman & Hall/CRC Mathematical & Computational Biology. *Chapman and Hall/CRC*, 6/2003.

[11] A. P. Gasch y M. B. Eisen: ***Exploring the conditional coregulation of yeast gene expression through fuzzy k-means clustering***. *Genome Biol.*, 3(11):research0059.1–research0059.22, 10/2002.

[12] M. B. Eisen, P. T. Spellman, P. O. Brown *y cols.*: ***Cluster analysis and display of genome-wide expression patterns***. *Proc. Natl. Acad. Sci. USA.*, 95(25):14863–14868, 12/1998.

[13] J. Quackenbush: ***Computational analysis of microarray data***. *Nat. Rev. Genet.*, 2(6):418–27, 6/2001.

[14] T. Kohonen: ***Self-Organizing Maps***. Information Sciences. *Springer-Verlag*, Berlin, 3ª ed., 2001.

[15] C. Garcia, M. Prieto y A. Pascual-Montano: ***A Speculative Parallel Algorithm for Self-Organizing Maps***. En G. R. Joubert, W. E. Nagel, F. J. Peters *y cols.* (eds.): ***Parallel***

Comput. Curr. Futur. Issues High End Comput., NIC, cap. 82, págs. 615–622. *John von Neumann Institute for Computing*, Jülich, 10/2006.

[16] P. Tamayo, D. Slonim, J. Mesirov *y cols.*: ***Interpreting patterns of gene expression with self-organizing maps: methods and application to hematopoietic differentiation***. *Proc. Natl. Acad. Sci. USA.*, 96(6):2907–12, 3/1999.

[17] P. Törönen, M. Kolehmainen, G. Wong *y cols.*: ***Analysis of gene expression data using self-organizing maps***. *FEBS Lett.*, 451(2):142–146, 5/1999.

[18] D. Jiang, C. Tang y A. Zhang: ***Cluster analysis for gene expression data: a survey***. *IEEE Trans. Knowl. Data Eng.*, 16(11):1370–1386, 11/2004.

[19] I. T. Jollife: ***Principal Component Analysis***. Springer Series in Statistics. *Springer*, New York, NY, 2ª ed., 2002.

[20] G. H. Golub y C. F. Van Loan: ***Matrix Computations***. *Johns Hopkins University Press*, Baltimore, 4ª ed., 12/2012.

[21] M. W. Berry, S. T. Dumais y T. A. Letsche: ***Computational Methods for Intelligent Information Access***. En ***Proc. ACM/IEEE Conf. Supercomput.***, pág. 20, New York, NY, 12/1995. San Diego Supercomputer Center, *ACM*.

[22] A. Hyvärinen, J. Karhunen y E. Oja: ***Independent Component Analysis***. Adaptive and Learning Systems for Signal Processing, Communications, and Control. *John Wiley & Sons, Inc.*, New York, NY, 5/2002.

[23] D. D. Lee y H. S. Seung: ***Learning the parts of objects by non-negative matrix factorization.*** *Nature*, 401(6755):788–791, 10/1999.

[24] S. Raychaudhuri, J. M. Stuart y R. B. Altman: ***Principal components analysis to summarize microarray experiments: application to sporulation time series***. En ***Pacific Symp. Biocomput.***, vol. **5**, págs. 455–66, 1/2000.

[25] B. W. Mel: ***Computational neuroscience. Think positive to find parts***. *Nature*, 401(6755):759–760, 10/1999.

[26] D. Guillamet y J. Vitrià: ***Non-negative matrix factorization for face recognition***. En M. Escrig, F. Toledo y E. Golobardes (eds.): ***Top. Artif. Intell.***, Lecture Notes in Computer Science, págs. 336–344. *Springer-Verlag*, Berlin, 10/2002.

[27] T. Feng, S. Li, H. Y. Shum *y cols.*: ***Local non-negative matrix factorization as a visual representation***. En ***Proc. 2nd Int. Conf. Dev. Learn.***, págs. 178–183, Los Alamitos, CA, 2002. *IEEE Computer Society*.

[28] R. Ramanath, W. E. Snyder y H. Qi: ***Eigenviews for object recognition in multispectral imaging systems***. En ***Proc. 32nd Appl. Imag. Pattern Recognit. Work.***, págs. 33–38, Los Alamitos, CA, 2003. *IEEE Computer Society*.

[29] X. Bai y C. Wang: ***Color face recognition based on NMF with Fisher rule***. En ***Proc. 2nd IEEE Int. Conf. Inf. Manag. Eng.***, págs. 467–471, Piscataway, NJ, 4/2010. *IEEE*.

[30] G. Buchsbaum y O. Bloch: ***Color categories revealed by non-negative matrix factorization of Munsell color spectra***. *Vision Res.*, 42(5):559–563, 3/2002.

[31] R. Ramanath, R. G. Kuehni, W. E. Snyder *y cols.*: ***Spectral spaces and color spaces***. *Color Res. Appl.*, 29(1):29–37, 2/2004.

[32] A. Heger y L. Holm: ***Sensitive pattern discovery with 'fuzzy' alignments of distantly related proteins***. *Bioinformatics*, 19(Suppl 1):i130–i137, 7/2003.

[33] P. M. Kim y B. Tidor: ***Subsystem identification through dimensionality reduction of large-scale gene expression data***. *Genome Res.*, 13(7):1706–18, 7/2003.

[34] J. P. Brunet, P. Tamayo, T. R. Golub *y cols.*: ***Metagenes and molecular pattern discovery using matrix factorization***. *Proc. Natl. Acad. Sci. USA.*, 101(12):4164–4169, 3/2004.

[35] P. Carmona-Saez, R. D. Pascual-Marqui, F. Tirado *y cols.*: ***Biclustering of gene expression data by Non-smooth Non-negative Matrix Factorization***. *BMC Bioinformatics*, 7:78, 2/2006.

[36] P. Tamayo, D. Scanfeld, B. L. Ebert *y cols.*: ***Metagene projection for cross-platform, cross-species characterization of global transcriptional states***. *Proc. Natl. Acad. Sci. USA.*, 104(14):5959–64, 4/2007.

[37] L. N. Hutchins, S. M. Murphy, P. Singh *y cols.*: ***Position-dependent motif characterization using non-negative matrix factorization.*** *Bioinformatics*, 24(23):2684–2690, 12/2008.

[38] W. Kong, X. Mou, Q. Li *y cols.*: ***Learning the local molecular pattern of Alzheimer's disease by non-negative matrix factorization***. En ***Proc. Int. Conf. Green Circuits Syst.***, págs. 621–625, Piscataway, NJ, 6/2010. *IEEE*.

[39] J. Arnedo, C. del Val, G. A. de Erausquin *y cols.*: ***PGMRA: a web server for (phenotype x genotype) many-to-many relation analysis in GWAS.*** *Nucleic Acids Res.*, 41(Web Server issue):W142–W149, 7/2013.

[40] H. Nakaoka, A. Tajima, T. Yoneyama *y cols.*: ***Gene expression profiling reveals distinct molecular signatures associated with the rupture of intracranial aneurysm.*** *Stroke.*, 45(8):2239–2245, 8/2014.

[41] S. A. Robila y L. G. Maciak: ***Sequential and Parallel Feature Extraction in Hyperspectral Data Using Nonnegative Matrix Factorization***. En ***Proc. IEEE Long Isl. Syst. Appl. Technol. Conf.***, págs. 1–7, Piscataway, NJ, 5/2007. *IEEE*.

[42] P. Smaragdis y J. Brown: ***Non-negative matrix factorization for polyphonic music transcription***. En ***Proc. IEEE Work. Appl. Signal Process. to Audio Acoust.***, págs. 177–180, Piscataway, NJ, 2003. *IEEE*.

[43] E. Battenberg y D. Wessel: ***Accelerating Non-Negative Matrix Factorization for Audio Source Separation on Multi-Core and Many-Core Architectures***. En K. Hirata, G. Tzanetakis y K. Yoshii (eds.): ***Proc. 10th Int. Soc. Music Inf. Retr. Conf.***, págs. 501–506. *International Society for Music Information Retrieval*, 2009.

[44] M. Chagoyen, P. Carmona-Saez, H. Shatkay *y cols.*: ***Discovering semantic features in the literature: a foundation for building functional associations***. *BMC Bioinformatics*, 7:41, 1/2006.

[45] K. Kanjani: ***Parallel non negative matrix factorization for document clustering***. Cpsc-659 (parallel and distributed numerical algorithms) course, Texas A&M University, 2007.

[46] M. Vazquez, P. Carmona-Saez, R. Nogales-Cadenas *y cols.*: ***SENT: semantic features in text.*** *Nucleic Acids Res.*, 37(Web Server issue):W153–W159, 7/2009.

[47] V. Kysenko, K. Rupp, O. Marchenko *y cols.*: ***GPU-Accelerated non-negative matrix factorization for text mining***. En G. Bouma, A. Ittoo, E. Métais *y cols.* (eds.): ***Nat. Lang. Process. Inf. Syst.***, Lecture Notes in Computer Science, págs. 158–163. *Springer-Verlag*, Berlin, 2012.

[48] W. Liu, N. Zheng y X. Lu: ***Non-negative matrix factorization for visual coding***. En ***Proc. IEEE Int. Conf. Acoust. Speech, Signal Process.***, vol. **3**, págs. 293–296, Piscataway, NJ, 2003. *IEEE*.

[49] A. Cichocki, R. Zdunek y S. i. Amari: ***New Algorithms for Non-Negative Matrix Factorization in Applications to Blind Source Separation***. En ***Proc. IEEE Int. Conf. Acoust. Speed Signal Process.***, vol. **5**, págs. V–621–V–624, Piscataway, NJ, 5/2006. *IEEE*.

[50] K. W. Wilson, B. Raj, P. Smaragdis *y cols.*: ***Speech denoising using nonnegative matrix factorization with priors***. En ***Proc. IEEE Int. Conf. Acoust. Speech Signal Process.***, págs. 4029–4032, Piscataway, NJ, 3/2008. *IEEE*.

[51] D. D. Lee y H. S. Seung: ***Algorithms for Non-negative Matrix Factorization***. En T. K. Leen, T. G. Dietterich y V. Tresp (eds.): ***Adv. Neural Inf. Process. Syst. 13 (NIPS 2000)***, págs. 556–562. *The MIT Press*, Cambridge, MA, 5/2001.

[52] Z. Chen, A. Cichocki y T. Rutkowski: ***Constrained non-Negative Matrix Factorization Method for EEG Analysis in Early Detection of Alzheimer Disease***. En ***Proc. IEEE Int. Conf. Acoust. Speed Signal Process.***, vol. **5**, págs. V–893–V–896, Piscataway, NJ, 5/2006. *IEEE*.

[53] A. Cichocki y R. Zdunek: ***Regularized Alternating Least Squares Algorithms for Non-negative Matrix/Tensor Factorization***. En D. Liu, S. Fei, Z. G. Hou *y cols.* (eds.): ***Adv. Neural Networks (ISNN 2007)***, Lecture Notes in Computer Science, págs. 793–802. *Springer-Verlag*, Berlin, 2007.

[54] A. Cichocki, R. Zdunek y S. i. Amari: ***Hierarchical ALS Algorithms for Nonnegative Matrix and 3D Tensor Factorization***. En M. E. Davies, C. J. James, S. A. Abdallah *y cols.* (eds.): ***Indep. Compon. Anal. Signal Sep.***, vol. **4666** de ***Lecture Notes in Computer Science***, págs. 169–176. *Springer-Verlag*, Berlin, 2007.

[55] V. Nikulin, T. H. Huang, S. K. Ng *y cols.*: ***A Very Fast Algorithm for Matrix Factorization***. *Stat. Probab. Lett.*, 81(7):773–782, 7/2011.

[56] S. Kullback y R. A. Leibler: ***On Information and Sufficiency***. *Ann. Math. Stat.*, 22(1):79–86, 3/1951.

[57] A. Pascual-Montano, J. M. Carazo, K. Kochi *y cols.*: ***Nonsmooth nonnegative matrix factorization (nsNMF)***. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(3):403–15, 3/2006.

[58] P. O. Hoyer: ***Non-negative sparse coding***. En ***Proc. 12th IEEE Work. Neural Networks Signal Process.***, págs. 557–565, Piscataway, NJ, 2002. *IEEE*.

[59] P. O. Hoyer: ***Non-negative matrix factorization with sparseness constraints***. *J. Mach. Learn. Res.*, 5:1457–1469, 12/2004.

[60] D. Dueck, Q. D. Morris y B. J. Frey: ***Multi-way clustering of microarray data using probabilistic sparse matrix factorization***. *Bioinformatics*, 21 Suppl 1:i144–51, 6/2005.

[61] G. Wang, A. V. Kossenkov y M. F. Ochs: ***LS-NMF: a modified non-negative matrix factorization algorithm utilizing uncertainty estimates***. *BMC Bioinformatics*, 7(1):175, 1/2006.

[62] H. Kim y H. Park: ***Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis***. *Bioinformatics*, 23(12):1495–502, 6/2007.

[63] P. Pehkonen, G. Wong y P. Törönen: ***Theme discovery from gene lists for identification and viewing of multiple functional groups***. *BMC Bioinformatics*, 6:162, 1/2005.

[64] G. Lohmann, K. G. Volz y M. Ullsperger: ***Using non-negative matrix factorization for single-trial analysis of fMRI data.*** *Neuroimage*, 37(4):1148–1160, 10/2007.

[65] K. Devarajan: ***Nonnegative matrix factorization: an analytical and interpretive tool in computational biology***. *PLoS Comput. Biol.*, 4(7):e1000029, 7/2008.

[66] Y. Gao y G. Church: ***Improving molecular cancer class discovery through sparse non-negative matrix factorization***. *Bioinformatics*, 21(21):3970–3975, 11/2005.

[67] K. Inamura, T. Fujiwara, Y. Hoshida *y cols.*: ***Two subclasses of lung squamous cell carcinoma with different gene expression profiles and prognosis identified by hierarchical clustering and non-negative matrix factorization***. *Oncogene*, 24(47):7105–13, 10/2005.

[68] D. R. Carrasco, G. Tonon, Y. Huang *y cols.*: ***High-resolution genomic profiles define distinct clinico-pathogenetic subgroups of multiple myeloma patients.*** *Cancer Cell*, 9(4):313–25, 4/2006.

[69] X. Han: ***Cancer molecular pattern discovery by subspace consensus kernel classification***. En ***Proc. LSS Comput. Syst. Bioinforma. Conf.***, vol. **6**, págs. 55–65, 8/2007.

[70] P. A. Northcott, A. Korshunov, H. Witt *y cols.*: ***Medulloblastoma comprises four distinct molecular variants***. *J. Clin. Oncol.*, 29(11):1408–1414, 4/2011.

[71] Cancer Genome Atlas Research Network, C. Kandoth, N. Schultz *y cols.*: ***Integrated genomic characterization of endometrial carcinoma***. *Nature*, 497(7447):67–73, 5/2013.

[72] G. Getz, E. Levine y E. Domany: ***Coupled two-way clustering analysis of gene microarray data***. *Proc. Natl. Acad. Sci. USA.*, 97(22):12079–84, 10/2000.

[73] A. Mukhopadhyay, U. Maulik y S. Bandyopadhyay: ***A novel biclustering approach to association rule mining for predicting HIV-1-human protein interactions***. *PLoS One*, 7(4):e32289, 4/2012.

[74] F. Zhou, Q. Ma, G. Li *y cols.*: ***QServer: a biclustering server for prediction and assessment of co-expressed gene clusters.*** *PLoS One*, 7(3):e32660, 1/2012.

[75] B. Pontes, R. Giráldez y J. S. Aguilar-Ruiz: ***Biclustering on expression data: A review.*** *J. Biomed. Inform.*, 57:163–180, 10/2015.

[76] S. Chen, J. Liu y T. Zeng: ***Measuring the quality of linear patterns in biclusters.*** *Methods*, 83:18–27, 7/2015.

[77] S. C. Madeira y A. L. Oliveira: ***Biclustering algorithms for biological data analysis: a survey***. *IEEE/ACM Trans. Comput. Biol. Bioinforma.*, 1(1):24–45, 1/2004.

[78] M. Crescenzi y A. Giuliani: ***The main biological determinants of tumor line taxonomy elucidated by a principal component analysis of microarray data***. *FEBS Lett.*, 507(1):114–118, 10/2001.

[79] L. Liu, D. M. Hawkins, S. Ghosh *y cols.*: ***Robust singular value decomposition analysis of microarray data***. *Proc. Natl. Acad. Sci. USA.*, 100(23):13167–72, 11/2003.

[80] S. Monti, P. Tamayo, J. Mesirov *y cols.*: ***Consensus clustering: A resampling-based method for class discovery and visualization of gene expression microarray data***. *Mach. Learn.*, 52(1-2):91–118, 7/2003.

[81] R. R. Sokal y F. J. Rohlf: ***The Comparison of Dendrograms by Objective Methods***. *Taxon*, 11(2):33, 2/1962.

[82] A. Cichocki y R. Zdunek: ***NMFLAB - MATLAB Toolbox for Non-Negative Matrix Factorization***, 2006. http://www.bsp.brain.riken.jp/ICALAB/nmflab.html [04-08-2014].

[83] K. Devarajan y G. Wang: ***Parallel implementation of non-negative matrix algorithms using high-performance computing cluster***. Scientific report, Fox Chase Cancer Center, 2006.

[84] A. Pascual-Montano, P. Carmona-Saez, M. Chagoyen *y cols.*: ***bioNMF: a versatile tool for non-negative matrix factorization in biology***. *BMC Bioinformatics*, 7:366, 1/2006.

[85] S. A. Robila y L. G. Maciak: ***A parallel unmixing algorithm for hyperspectral images***. En D. P. Casasent, E. L. Hall y J. Röning (eds.): ***Intell. Robot. Comput. Vis. XXIV Algorithms, Tech. Act. Vis.***, SPIE Proceedings, págs. 63840F–63840F–11, 10/2006.

[86] R. Gaujoux y C. Seoighe: ***A flexible R package for nonnegative matrix factorization.*** *BMC Bioinformatics*, 11:367, 7/2010.

[87] C. Dong, H. Zhao y W. Wang: ***Parallel Nonnegative Matrix Factorization Algorithm on the Distributed Memory Platform***. *Int. J. Parallel Program.*, 38(2):117–137, 4/2010.

[88] N. Lopes y B. Ribeiro: ***Non-negative Matrix Factorization. Implementation using Graphics Processing Units***. En C. Fyfe, P. Tino, D. Charles *y cols.* (eds.): ***Intell. Data Eng. Autom. Learn. (IDEAL 2010)***, Lecture Notes in Computer Science, págs. 275–283. *Springer-Verlag*, Berlin, 2010.

[89] J. Platoš, P. Gajdoš, P. Krömer *y cols.*: ***Non-negative Matrix Factorization on GPU***. En F. Zavoral, J. Yaghob, P. Pichappan *y cols.* (eds.): ***Networked Digit. Technol.***, Communications in Computer and Information Science, págs. 21–30. *Springer-Verlag*, Berlin, 2010.

[90] K. J. Savage, S. Monti, J. L. Kutok *y cols.*: ***The molecular signature of mediastinal large B-cell lymphoma differs from that of other diffuse large B-cell lymphomas and shares features with classical Hodgkin lymphoma***. *Blood*, 102(12):3871–3879, 12/2003.

[91] Y. Gómez-Llorente, R. J. Fletcher, X. S. Chen *y cols.*: ***Polymorphism and Double Hexamer Structure in the Archaeal Minichromosome Maintenance (MCM) Helicase from Methanobacterium thermoautotrophicum***. *J. Biol. Chem.*, 280(49):40909–40915, 12/2005.

[92] The MathWorks Inc: ***MATLAB***. http://www.mathworks.com/products/matlab.

[93] Borland Corp: ***Delphi***. http://www.codegear.com/products/delphi [03-08-2014].

[94] R. C. Whaley y A. Petitet: ***Minimizing development and maintenance costs in supporting persistently optimized BLAS***. *Softw. Pract. Exp.*, 35(2):101–121, 2/2005.

[95] Intel Corp.: ***Math Kernel Library (MKL)***, 5/2003. https://software.intel.com/en-us/intel-mkl.

[96] Q. Wang, X. Zhang, Y. Zhang *y cols.*: ***AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs***. En ***Proc. Int. Conf. High Perform. Comput. Networking, Storage Anal. (SC '13)***, págs. 25:1–25:12, New York, NY, 11/2013. *ACM Press*.

[97] C. L. Lawson, R. J. Hanson, D. R. Kincaid *y cols.*: ***Basic Linear Algebra Subprograms for Fortran Usage***. *ACM Trans. Math. Softw.*, 5(3):308–323, 9/1979.

[98] NVIDIA Corp: ***CUDA: Compute Unified Device Architecture***. http://www.nvidia.com/object/cuda_home.html [29-07-2014].

[99] Khronos Group: ***OpenGL.org - The Industry's Foundation for High Performance Graphics***. http://www.opengl.org/ [05-08-2014].

[100] R. Fernando y M. J. Kilgard: ***The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics***. *Addison-Wesley Longman Publishing Co., Inc.*, Boston, MA, 3/2003.

[101] Khronos Group: ***OpenCL - The open standard for parallel programming of heterogeneous systems***, 2008. https://www.khronos.org/opencl/ [03-11-2015].

[102] M. Harris y D. Göddeke: ***GPGPU.org: General-Purpose Computation on Graphics Hardware***, 2004. http://gpgpu.org [04-08-2014].

[103] H. group: ***hgpu.org: high performance computing on graphics processing units***, 2010. http://hgpu.org [15-03-2015].

[104] ***GPUComputing.net***, 11/2009. http://www.gpucomputing.net [13-09-2015].

[105] J. Nickolls y W. J. Dally: ***The GPU Computing Era***. *IEEE Micro*, 30(2):56–69, 3/2010.

[106] J. D. Owens, D. Luebke, N. Govindaraju *y cols.*: ***A Survey of General-Purpose Computation on Graphics Hardware***. *Comput. Graph. Forum*, 26(1):80–113, 3/2007.

[107] J. Setoain, M. Prieto, C. Tenllado *y cols.*: ***GPU for Parallel On-Board Hyperspectral Image Processing***. *Int. J. High Perform. Comput. Appl.*, 22(24):424–437, 11/2008.

[108] N. Govindaraju, J. Gray, R. Kumar *y cols.*: ***GPUTeraSort: high performance graphics co-processor sorting for large database management***. En ***Proc. ACM SIGMOD Int. Conf. Manag. data***, págs. 325–336. *ACM Press*, 6/2006.

[109] C. M. Isborn, N. Luehr, I. S. Ufimtsev *y cols.*: ***Excited-State Electronic Structure with Configuration Interaction Singles and Tamm-Dancoff Time-Dependent Density Functional Theory on Graphical Processing Units.*** *J. Chem. Theory Comput.*, 7(6):1814–1823, 6/2011.

[110] H. Burau, R. Widera, W. Honig *y cols.*: ***PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster***. *IEEE Trans. Plasma Sci.*, 38(10):2831–2839, 10/2010.

[111] D. G. McArt, P. Bankhead, P. D. Dunne *y cols.*: ***cudaMap: a GPU accelerated program for gene expression connectivity mapping.*** *BMC Bioinformatics*, 14:305, 1/2013.

[112] M. C. Schatz, C. Trapnell, A. L. Delcher *y cols.*: ***High-throughput sequence alignment using Graphics Processing Units.*** *BMC Bioinformatics*, 8:474, 1/2007.

[113] A. W. Ghoorah, M. D. Devignes, M. Smaïl-Tabbone *y cols.*: ***Protein docking using case-based reasoning.*** *Proteins Struct. Funct. Bioinforma.*, 81(12):2150–2158, 10/2013.

[114] P. Richmond, D. Walker, S. Coakley *y cols.*: ***High performance cellular level agent-based simulation with FLAME for the GPU.*** *Brief. Bioinform.*, 11(3):334–347, 5/2010.

[115] L. Dematté y D. Prandi: ***GPU computing for systems biology.*** *Brief. Bioinform.*, 11(3):323–333, 5/2010.

[116] C. Jiang y M. Snir: ***Automatic tuning matrix multiplication performance on graphics hardware***. En ***Proc. 14th Int. Conf. Parallel Archit. Compil. Tech.***, págs. 185–194, Los Alamitos, CA, 2005. *IEEE Computer Society*.

[117] N. Galoppo, N. Govindaraju, M. Henson *y cols.*: ***LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware***. En ***Proc. ACM/IEEE Supercomput. Conf.***, pág. 3, Los Alamitos, CA, 11/2005. *IEEE Computer Society*.

[118] B. Oancea y T. Andrei: ***Developing a High Performance Software Library with MPI and CUDA for Matrix Computations***. *Comput. Methods Soc. Sci.*, 1(2):5–10, 12/2013.

[119] S. Rixner, W. J. Dally, U. J. Kapasi *y cols.*: ***A Bandwidth-efficient Architecture for Media Processing***. En ***Proc. 31st Annu. ACM/IEEE Int. Symp. Microarchitecture***, págs. 3–13, Los Alamitos, CA, 1998. *IEEE Computer Society*.

[120] M. B. Taylor, J. Kim, J. Miller *y cols.*: ***The Raw microprocessor: a computational fabric for software circuits and general-purpose programs***. *IEEE Micro*, 22(2):25–35, 3/2002.

[121] H. Hofstee: ***Power Efficient Processor Architecture and The Cell Processor***. En ***Proc. 11th Int. Symp. High-Performance Comput. Archit.***, págs. 258–262, Los Alamitos, CA, 2005. *IEEE Computer Society*.

[122] J. Gummaraju y M. Rosenblum: ***Stream Programming on General-Purpose Processors***. En ***Proc. 38th Annu. IEEE/ACM Int. Symp. Microarchitecture***, págs. 343–354, Los Alamitos, CA, 11/2005. *IEEE Computer Society*.

[123] B. Khailany, W. J. Dally, U. J. Kapasi *y cols.*: ***Imagine: Media Processing with Streams***. *IEEE Micro*, 21(2):35–46, 3/2001.

[124] I. Buck, T. Foley, D. Horn *y cols.*: ***Brook for GPUs***. *ACM Trans. Graph.*, 23(3):777–786, 8/2004.

[125] J. Setoain, C. Tenllado, M. Prieto *y cols.*: ***Parallel Hyperspectral Image Processing on Commodity Graphics Hardware***. En ***Proc. Int. Conf. Parallel Process. Work.***, págs. 465–472, Los Alamitos, CA, 8/2006. *IEEE Computer Society*.

[126] R. Edgar: ***Gene Expression Omnibus: NCBI gene expression and hybridization array data repository***. *Nucleic Acids Res.*, 30(1):207–210, 1/2002.

[127] G. Rustici, N. Kolesnikov, M. Brandizi *y cols.*: ***ArrayExpress update–trends in database growth and links to data analysis tools***. *Nucleic Acids Res.*, 41(Database issue):D987–D990, 1/2013.

[128] I. Foster y C. Kesselman (eds.): ***The Grid: Blueprint for a New Computing Infrastructure***. *Morgan Kaufmann Publishers*, San Francisco (CA), 1ª ed., 8/1998.

[129] I. Foster, C. Kesselman y S. Tuecke: ***The Anatomy of the Grid - Enabling Scalable Virtual Organizations***. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 8/2001.

[130] I. Foster: ***What is the Grid? A Three Point Checklist***. *GRIDToday*, 1(6), 7/2002.

[131] I. Foster: ***Globus toolkit version 4: Software for service-oriented systems***. En H. Jin, D. Reed y W. Jiang (eds.): ***Netw. Parallel Comput.***, Lecture Notes in Computer Science, págs. 2–13. *Springer-Verlag*, Berlin, 2005.

[132] M. Romberg: ***The UNICORE Grid infrastructure***. *Sci. Program.*, 10(2):149–157, 7/2002.

[133] M. Ellert, M. Gronager, A. Konstantinov *y cols.*: ***Advanced Resource Connector middleware for lightweight computational Grids***. *Futur. Gener. Comput. Syst.*, 23(2):219–240, 2/2007.

[134] C. Aiftimiei, A. Aimar, A. Ceccanti *y cols.*: ***Towards next generations of software for distributed infrastructures: The European Middleware Initiative***. En ***Proc. 8th IEEE Int. Conf. E-Science***, págs. 1–10, Piscataway, NJ, 10/2012. *IEEE*.

[135] Enabling Grids for E-sciencE (EGEE) Project: ***gLite: Lightweight Middleware for Grid Computing***, 5/2006. http://glite.cern.ch [03-08-2014].

[136] ***OpenPBS.org***. http://www.openpbs.org [03-08-2014].

[137] Sun Microsystems Inc (acquired by Oracle Corp): ***SGE: Sun Grid Engine***, 1994. http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html.

[138] D. Thain, T. Tannenbaum y M. Livny: ***Distributed computing in practice: the Condor experience***. *Concurr. Comput. Pract. Exp.*, 17(2-4):323–356, 2/2005.

[139] I. M. Carrión, E. Huedo y I. M. Llorente: ***Interoperating grid infrastructures with the GridWay metascheduler***. *Concurr. Comput. Pract. Exp.*, 27(9):2278–2290, 6/2015.

[140] C. Smith: ***Open source metascheduling for virtual organizations with the community scheduler framework (CSF)***. *Tech. whitepaper, Platf. Comput.*, 2003.

[141] K. Kurowski, B. Ludwiczak, J. Nabrzyski *y cols.*: ***Dynamic Grid Scheduling with Job Migration and Rescheduling in the GridLab Resource Management System***. *Sci. Program.*, 12(4):263–273, 2004.

[142] R. Liao, Y. Zhang, J. Guan *y cols.*: ***CloudNMF: a MapReduce implementation of non-negative matrix factorization for large-scale biological datasets.*** *Genomics. Proteomics Bioinformatics*, 12(1):48–51, 2/2014.